hyväksymispäivä arvosana

arvostelija

Tietoliikenteen ruuhkanhallinta datakeskuksissa

Matti Ruoho

Helsinki 29.4.2017

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta – Fakultet –Faculty	Laitos – Institution – Department
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä – Författare – Author

Matti Ruoho

Työn nimi – Arbetets titel – Title

Tietoliikenteen ruuhkanhallinta datakeskuksissa

Oppiaine – Läroämne – Subject

Tietojenkäsittelytiede

Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
Pro-gradu	29.4.2017	58 sivua + 5 liitesivua

Tiivistelmä – Referat – Abstract

Datakeskukset koostuvat suuresta joukosta palvelimia jotka tallentavat ja käsittelevät suuria datamääriä. Niiden avulla on mahdollista toteuttaa suuria laskennallisia operaatioita hajauttamalla ne tehokkaasti eri palvelimien kesken, laskemalla yhteen eri palvelimien palauttamat tulokset ja palauttamalla lopputulos käyttäjälle tai palvelun pyytäjälle. Esimerkkejä tästä ovat Googlen ja Yahoon kaltaisten yhtiöiden käyttämät ohjelmistot ja algoritmit kuten MapReduce ja Search.

Datakeskukset ovat monin osin erilaisia verrattuna Internetiin, esimerkiksi kun puhutaan verkkojen topologisesta rakenteesta, siirtonopeuksista ja muista verkkoliikenteen ominaisuuksista, minkä vuoksi datakeskuksista on tullut merkittävä tutkimuskohde. Koska erot Internetiin ovat huomattavia, datakeskuksissa joudutaan tekemään toisenlaisia ratkaisuja tietoliikenteen ruuhkautumisen välttämiseksi. Internetin kuljetusprotokolla TCP ei sellaisenaan sovi datakeskuksen kaltaiseen ympäristöön; tämän vuoksi TCP:stä on jouduttu kehittämään datakeskuksia varten erilaisia muunnoksia.

Ohjelmistot, joita ajetaan datakeskuksissa, voidaan pääasiassa jakaa kahteen kategoriaan kun vertaillaan millaista niiden tietoliikenne on: 1) Toisille on ominaista korkea läpisyöttö jossa lähetetään suurikokoisia viestejä, toisin sanoen verkon linkit ovat jatkuvasti puskuroitu täyteen siirrettävästä datasta; 2) toisenlaisissa sovelluksissa on tärkeää että tieto siirtyy ilman viivytyksiä (toisin sanoen vastausajat ovat pieniä) ja siirrettävät viestit ovat näin ollen pienempiä. TCP Incast on tilanne jossa suuri joukko palvelimia lähettää tietoa yhdelle käyttäjälle samanaikaisesti ja dataa on niin paljon ettei vastaanottaja pysty puskuroimaan kaikkea dataa; tämä johtaa datapakettien häviämiseen ja uudelleen lähettämiseen. TCP Outcast on edellistä vastakkainen tilanne jossa suuret tietovirrat ja pienemmät tietovirrat ovat matkalla samaan aikaan samasta verkon portista ulos ja pienemmät tietovirrat joutuvat jonottamaan isompien tietovirtojen takana.

Kun toteutetaan tiedonsiirtoa datakeskuksessa, joudutaan tekemään kompromissi sen välillä minkä tyyppistä tietoliikennettä datakeskuksessa ajettavat ohjelmistot harjoittavat. Siksi haasteena onkin toteuttaa yleinen protokolla joka pystyy mukautumaan erilaisiin tilanteisiin ilman että verkon laitteistoon tarvitsisi tehdä muutoksia.

Tässä tukielmassa luodaan yleiskatsaus siihen miten datakeskusten ruuhkanhallintaa on tutkittu viime vuosina. Ruuhkan syitä esitellään käymällä läpi datakeskusten topologioiden erityisiä ominaisuuksia ja sitä kautta datakeskusten ja Internetin tietoliikenteen välisiä eroavaisuuksia. Eri ruuhkanhallinta-algoritmeja esitellään ja lopuksi suoritetaan vertailu jossa eri algoritmien etuja ja heikkouksia pohditaan.

ACM Computing Classification System (CCS):

• Networks~Data center networks • Networks~Transport protocols

Avainsanat – Nyckelord – Keywords

Datakeskus, ruuhkanhallinta, TCP

Säilytyspaikka – Förvaringställe – Where deposited

Muita tietoja – Övriga uppgifter – Additional information

Sisältö

1 Johdanto

Datakeskuksissa suoritettavat ohjelmistot liittyvät usein pilvilaskentaan, sosiaaliseen mediaan ja informaatioverkostoihin joiden tarjoamia palveluita ovat esimerkiksi hakukoneet ja uutiset. Tällaisissa ohjelmistoissa laskenta hajautetaan lukuisten palvelimien kesken joita voi olla sadoista tuhansiin. GFS, BigTable, MapReduce ja Dryad ovat esimerkkejä tällaisista ohjelmistoista joissa monien koneiden välillä liikkuu suuria määriä dataa.

TCP on Internetin kuljetuskerroksen protokolla joka ei sellaisenaan sovi datakeskusten kaltaisten ympäristöjen tarpeisiin joille tyypillistä ovat pienet puskurimuistit kytkimissä, nopeat linkit sekä pienet viiveet tiedonsiirrossa. Datakeskuksissa ajettavat ohjelmat generoivat pääasiassa kahden tyyppisiä datavirtoja (data flow) joita ovat lyhyet viestit joille on asetettu kriittiset aikarajat, sekä suuret viestit jotka vievät suuren osan kaistasta (bandwidth, esim. päivitykset) ja joiden lähetys kestää pitkään.

TCP:n heikkouksia datakeskuksissa ovat että se tarvitsee suuria kytkinten puskurimuisteja (datakeskuksissa kytkinten puskurimuistit ovat yleensä pieniä taloudellisista syistä), se käyttää kytkinten puskurimuistia raskaasti ja se ei suoriudu hyvin tietokantahakujen kaltaisista operaatioista jotka aiheuttavat ‘räjähdysmäisiä’, lyhyitä datavirtoja.

Tutkielman rakenne on seuraava: johdannon jälkeisessä osuudessa esitellään datakeskusverkkojen yleisiä ominaisuuksia ja esitellään muutama esimerkki datakeskuksissa käytetyistä topologioista. Kolmannessa osiossa määritellään datakeskusten ruuhkaantumisen syyt ja ruuhkan eri tyyppejä. Neljännessä osiossa puhutaan Internetin kaltaisissa verkoissa käytetystä ruuhkanhallinnasta jota ovat pääasiassa TCP monine variaatioineen. Osiossa 5 esitellään ruuhkanhallinta-algoritmeja jotka on suunniteltu nimenomaan datakeskusten käyttöön. Luvussa 6 vertaillaan datakeskusten ruuhkanhallinta-algoritmeja. Lopuksi yhteenvedossa (luvussa 7) esitellään tutkimuksen johtopäätökset.

2 Datakeskukset

Datakeskusten arkkitehtuurin voidaan karkeasti määritellä koostuvan neljästä komponentista: fyysisestä topologiasta, topologian reitityksestä, polkujen valinnasta

sekä ruuhkanhallinnasta [RPB10]. Nämä komponentit eivät sulje toisiaan pois vaan yhden suorituskyky vaikuttaa muiden suorituskykyyn.

Perinteisesti datakeskusten topologia on ollut hierarkkinen ja siinä on kaksi tai kolme kerrosta jossa on juurisolmut, lehtisolmut (joilla ei ole lapsisolmuja) sekä näiden kahden edellisen välissä olevat sisäolmut. Kirjallisuudessa käytetään yleisesti datakeskuksen topologian kerroksista englanninkielisiä nimityksiä aggregate- (sisä-), access- (lehti-) sekä core (juuri-). Tässä tekstissä kun jatkossa puhutaan eri kerrosten kytkimistä, käytetään nimityksiä juurikytkimet (= juurisolmut), sisäkytkimet (= sisäsolmut) ja lehtikytkimet (= lehtisolmut).

Datakeskus sisältää reitittimiä ja kytkimiä joko kahden tai kolmen kerroksen hierarkiassa. Lehtikerroksen kytkimet liittyvät sisäkerroksen kytkimeen ja nämä kytkimet liittyvät lopulta hierarkian juuressa olevaan juurikytkimeen. Kahden kerroksen hierarkiassa ei ole ollenkaan sisäkytkimiä ja kolmen kerroksen hierarkia sisältää kaikkien kolmen kerroksen kytkimiä. Palvelimet yhdistyvät datakeskukseen lehtikytkinten kautta ja palvelimet ovat ryhmitelty erillisiin räkkeihin ja yhdessä räkissä on tyypillisesti muutama kymmentä palvelinta.

Hierarkkinen topologia on toimiva ratkaisu jos suurin osa tietoliikenteestä virtaa sisään tai ulos datakeskuksesta. Pilvilaskennalle on yleistä että data liikkuu räkkien välillä mutta yritysten ja yliopistojen kaltaisissa verkoissa data kiertää koko verkon läpi. Jos suurin osa liikenteestä liikkuu datakeskuksen sisällä, topologian korkeimmista tasoista tulee hankalia pullonkauloja tietoliikenteelle. Tämän ongelman ratkaisemiseksi on kehitetty esimerkiksi FatTree-topologia joka käyttää useampia juurikytkimiä. Viime aikoina datakeskukset on yleensä rakennettu niin että niissä on monta juurikytkintä ja monta kerrosta. Ihanteellisesti datakeskuksessa tulisi olla täysin vapaa kaista jokaisen kahden palvelimen välillä (bisection bandwidth), toisin sanoen jokaisen palvelimen tulisi pystyä kommunikoimaan minkä tahansa muun palvelimen kanssa suurimmalla mahdollisella nopeudella.

Koska datakeskusten koot kasvavat ja samalla myös niille asetetut toiminnalliset vaatimukset, datakeskusten toteuttamisessa on erilaisia haasteita kuten liian pienet kytkinten puskurit (switch buffer), rajoittunut kaistan (bandwidth) käyttö, ruuhkanhallinta ja tietoturvaan liittyvät seikat. Alati kasvavat datakeskukset voivat yhdistää tuhansia palvelimia keskenään tukeakseen erilaisia laskennallisia tehtäviä

kuten pilvipalveluita ja tämän vuoksi tehokkaiden ja skaalautuvien topologioiden tutkiminen on merkittävä tutkimusalue.

Koska ideaalisesti datakeskus pitäisi pystyä toteuttamaan halvalla laitteistolla, tämä asettaa datakeskuksen hierarkian suunnittelussa vastakkain taloudelliset ja operatiiviset kustannukset; halvempaa laitteistoa joudutaan todennäköisemmin korjaamaan useammin, ja jos datakeskus on suunniteltu modulaariseksi eli vapaasti paikasta toiseen siirrettäväksi (esim. sijoitetaan palvelimet ym. konttiin), korjaamisesta voi tulla käytännössä hankalaa. Tämän vuoksi edullisemman datakeskuksen kestävyyteen pitäisi erityisesti panostaa, toisin sanoen datakeskuksen tulee kestää mahdollisimman kauan ennen kuin huoltotoimenpiteisiin tarvitsee ryhtyä. Pilvipalveluja ajavat datakeskukset eroavat monelta osin yritysten yksityiseen käyttöön tarkoitetuista datakeskuksista. Yrityksien käyttämät datakeskukset vaativat enemmän henkilökuntaa ylläpitoon kun taas pilvipalveluihin suunnatut datakeskukset perustuvat pitkälti automaatioon. Yritysten datakeskuksissa laitteisto koostuu yleensä kalliista korkeatasoisista palvelimista ja kytkimistä kun taas pilvipalvelujen kohdalla laitteisto on halvempaa ja pilvipalveluilta edellytetään parempaa skaalautuvuutta. Tässä tutkielmassa käsitellyt ruuhkanhallinta-algoritmit ja topologiat ovat paljolti suunniteltu pilvipalvelujen datakeskuksia ajatellen. Yritysten ja pilvipalveluiden datakeskusten lisäksi voidaan luokitella kolmas ryhmä joka on yliopistokampuksien käyttämät datakeskukset [SJ15].

Puumainen kolmikerroksinen puu (three-tier) on yleisin käytössä oleva topologia datakeskuksissa mutta (pääasiassa) taloudellisten kustannuksen vuoksi sille on etsitty seuraajaa. FatTree, DCell ja BCube ovat esimerkkejä topologioista jotka saattavat korvata kolmikerroksisen puun tulevaisuudessa. FatTree, kolmikerroksinen puu sekä BCube esitellään myöhemmin omissa aliluvuissaan.

Datakeskukset voidaan topologiansa osalta jakaa kahteen luokkaan [ZBC16]. Kytkinkeskeisissä (switch-based) topologioissa kytkimet hoitavat datan reititykseen liittyvän logiikan ja jokainen palvelin on yhteydessä muuhun verkkoon yhden portin kautta. Esimerkkejä kytkinkeskeisistä topologioista ovat edellä mainitut kolmikerroksinen puu ja FatTree. Palvelinkeskeisissä (server-based) topologioissa palvelimet itse hoitavat reitityksen ja nämä palvelimet liittyvät muuhun verkkoon useiden porttien kautta ja verkossa kytkimet ovat vain välipysäkkejä. Esimerkkejä palvelinkeskeisistä arkkitehtuureista ovat BCube ja DCell jotka soveltuvat sellaisten

sovellusten käyttöön jotka vaativat runsaasti kaistaa.. Palvelinkeskeiset topologiat luovat verkon rekursiivisesti luomalla perusyksikön ja laajentamalla sen suurempiin verkkoihin. Tämä perusyksikkö koostuu *n* palvelimesta jotka kaikki liittyvät yhteen *n*-porttiseen kytkimeen.

Bradjonic et al. [BSW14] vertailevat FatTree-, Hypergrid- ja satunnaisverkko (random graph)-topologioita ja tekevät johtopäätöksiä jokaisen hyvistä ja huonoista puolista, liittyen ruuhkautumiseen, luotettavuuteen ja taloudellisiin kustannuksiin. Tulosten perusteella FatTreessa on vähiten ruuhkaa (perustuen räkkien välillä virtaavaan datamäärään) ja se on hyvin luotettava verkko mutta käyttää runsaasti portteja. Koska FatTree vaatii paljon portteja käyttöönsä se tuottaa vakavia ongelmia tulevaisuudessa datakeskusten skaalautuvuuden kannalta datakeskusten yhä laajentuessa ja näin palvelimien määrän kasvaessa. Satunnaisverkkojen kohdalla kirjoittajat havaitsevat ruuhkautumisen olevan pientä ja satunnaisverkko tarvitsee vähemmän portteja mutta ei ole yhtä luotettava kuin FatTree. Hypergridissa havaitaan porttien määrän olevan vakio mihin tahansa datakeskuksen kokoon nähden mutta Hypergridissa räkkien välinen liikenne on vähäisempää kuin FatTreessa ja satunnaisverkossa.

De Souza Couto et al. [SSC16] vertailevat eri topologioita keskittyen siihen kuinka kukin pärjää laitteistovikojen ilmetessä. Kirjoittajat vertailevat yleistä kolmikerroksista puuta, FatTreeta, BCubea ja DCellia käyttäen pääasiallisena mittarina sitä kuinka kauan keskimäärin kestää datakeskuksen käyttöönotosta siihen verkon ensimmäinen palvelin hajoaa. Tulosten perusteella kolmikerroksinen puu häviää kestävyydessä muille vertailussa oleville topologioille mutta toisaalta jos se käyttää verkon sisä- ja juurikerroksessa luotettavaa kalustoa, kolmikerroksinen puu on yhtä kestävä kuin FatTree. Saman tutkimuksen perusteella FatTree on kestävä juurikerroksessa mutta lehtikerroksessa se häviää kestävyydessä BCubelle sekä DCellille. Toisaalta kolmikerroksinen puu ja FatTree pystyvät säilyttämään polkujen pituuden BCubeen ja DCelliin verrattuna laitteistovahinkojen ilmetessä; jos BCuben ja DCellin kohdalla tapahtuu kytkinten kaatumista, niissä keskimääräinen polun pituus kahden palvelimen välillä kasvaa enemmän kuin kolmikerroksisessa puussa ja FatTreessa. BCube selviää parhaiten verkon linkkien hajotessa; 84% verkon palvelimista pysyy ylhäällä jos 40% linkeistä on rikki (DCellin tapauksessa 74% palvelimista pysyy toimivana). Tämän voi selittää, sen lisäksi että BCuben suunnittelussa kestävyyteen on panostettu, sillä että

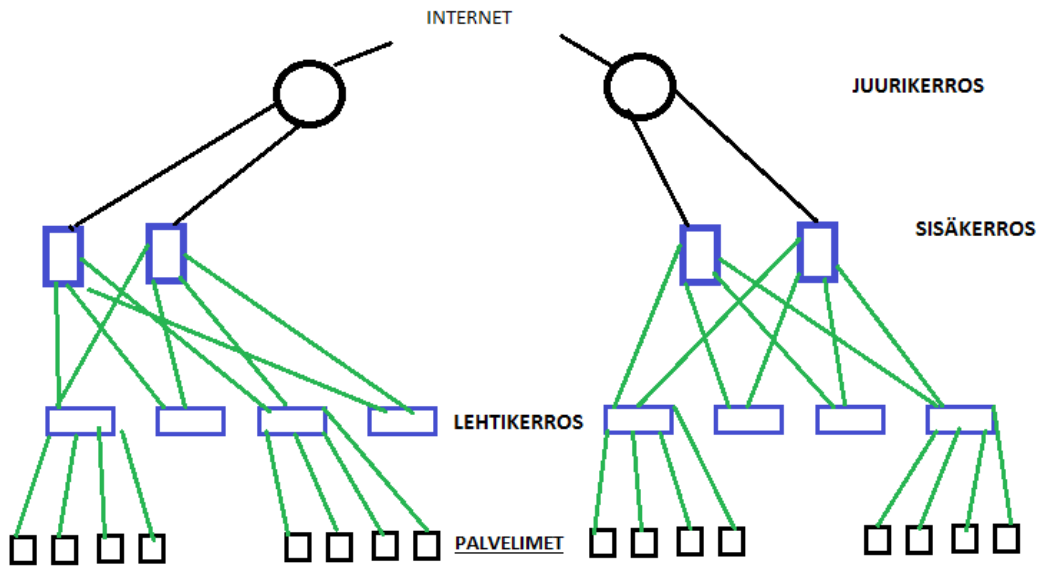
BCubessa palvelimet ovat suoraan yhteydessä ainoastaan kytkimiin; DCellin tapauksessa palvelimet ovat suoraan linkitetty toisiinsa. BCuben tapauksessa jos verkon linkki hajoaa, koko verkon kaatuminen on epätodennäköisempää kuin DCellissa.

Samassa tutkimuksessa havaitaan että DCell suhtautuu parhaiten kytkinten hajoamiseen. Tämä selittyy sillä että DCellissa palvelimilla on suurempi vastuu pitää verkko toimivana.

2.1 Kolmikerroksinen puu

Kolmikerroksinen puu (kuva 2.1) on eniten käytetty topologia datakeskuksissa [ZBC16]. Se on kytkimistä ja reitittimistä koostuva puu jossa on juuri-, sisä- ja lehtikerros. Lehtikerroksen kytkimissä on joukko 1 GigE portteja ja sisäkerroksen kytkimien kanssa kommunikoimiseen joukko 10 GigE portteja. Verkon palvelimilla on 1 GigE linkki lehtikytkimiin jotka ovat halvempia Ethernet-kytkimiä jotka yhdistävät samassa räkissä olevia palvelimia 1 GigE linkkien välityksellä. Sisäkytkimet yhdistyvät juurikytkimiin ja juurikerroksessa on juurikytkinten lisäksi yksi tai useampi reititin joka yhdistää datakeskuksen Internetiin tai muihin datakeskuksiin. Sisäkerroksessa on yleensä load balancer.

Topologian suurin haittapuoli on kalliit 10 GigE kytkimet sisäkerroksella. Muita heikkouksia ovat suuret reititystaulut jonka takia hakukomennot ovat pitkiä sekä oversubscriptioniin liittyvät seikat. Seuraavaksi esitelty FatTree on rakennettu



Kuva 2.1: Kolmikerroksinen puu [ZBC16].

edellämainittuja puutteita silmällä pitäen.

2.2 Fat-tree

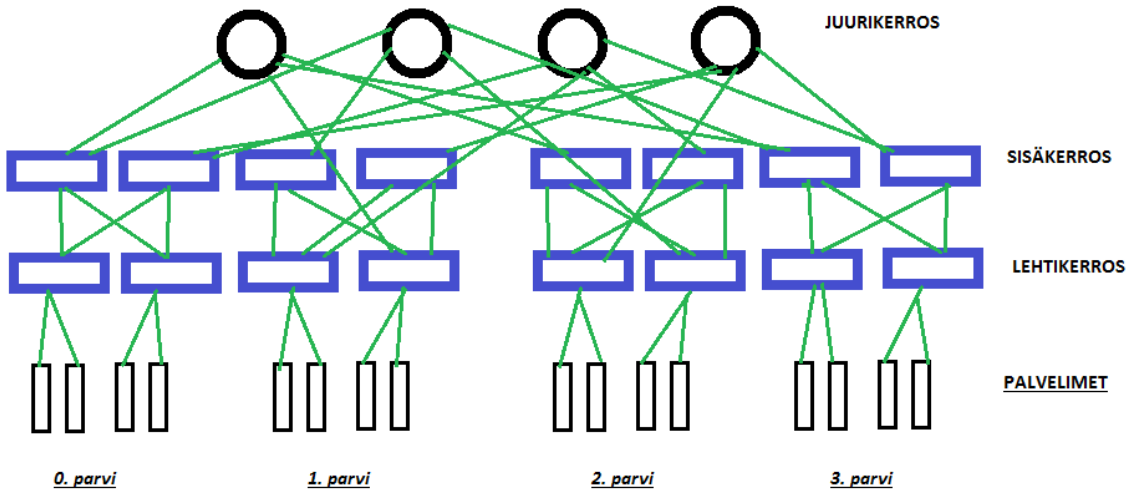
Tavalliseen puurakenteeseen verrattuna FatTreessa on lisätty kytkimiä juurikerrokseen ja kaikki sisä- ja lehtikytkimet jaetaan erillisiin parviin (pod). Kuva 2.2) esittelee *k*-FatTreen [ZBC16] missä k =4. Puussa on k = 4 parvea ja jokaisessa parvessa on sekä sisä- että juurikerroksen kytkimiä ja palvelimia.

Jokainen sisäkytkin yhdistyy kahteen (k / 2) juurikytkimeen jonka ansiosta jokainen juurikytkin pystyy kommunikoimaan jokaisen parven kanssa; juurikytkimiä on 4 ((k/2)2) kpl ja jokaisesta juurikytkimestä on jokaiseen parveen yksi portti. Yleisesti *k*-FatTree, jossa yhdelllä kytkimellä on k porttia, pystyy ylläpitämään k3 / 4 palvelinta. Tyypillisessä datakeskuksessa käytettävässä FatTreessa on 2 tai 3 kerrosta. Jokaisella verkon linkillä on sama tiedonsiirtonopeus.

FatTree voi käyttää yksinomaan halpoja kytkimiä jokaisessa kerroksessa minkä vuoksi FatTree on huomattavasti halvempi toteuttaa kuin kolmikerroksinen puu. Bradonjic et al. mainitsevat että FatTree saattaa saavuttaa rajansa lähitulevaisuudessa johtuen siitä miten rajoittuneesti lehtikytkimissä voi olla portteja.

2.3 Bcube

BCube [GLL09] suunniteltiin moduulaaristen datakeskusten käyttöön joille on tärkeää pitkä kestävyys t.s. niiden on tarkoitus kestää mahdollisimman pitkään ilman



Kuva 2.2: FatTree [ZBC16].

laitteistovikoja koska niiden huoltotoimenpiteet voivat olla hankalia. BCube on palvelinkeskeinen topologia jossa reitityslogiikka on palvelimien vastuulla. BCube on rekursiivisesti rakennettu koostuen tasoista joissa on halpoja kytkimiä. Perusyksikkö BCubessa on $BCube_0$ joka koostuu yhdestä kytkimestä jossa on *n* porttia *n* palvelimeen. $BCube_1$ (kuva 2.3.1) sen sijaan koostuu *n*:stä $BCube_0$:sta ja *n*:stä kytkimestä. Jokainen kytkin on yhteydessä jokaiseen $BCube_0$:aan jokaisen $BCube_0$:n yhden palvelimen kautta.

Rekursiivisesti $BCube_k$ koostuu *n*:stä $BCube_{k-1}$:sta ja n^k kytkimestä joissa jokaisessa on *n* porttia. $BCube_k$:ssa on n^{k+1} palvelinta ja $k+1$ tasoa ja jokaisella tasolla on n^k *n*-porttista kytkintä. $BCube_k$:n rakentamiseksi kaikki *n* $BCube_{k-1}$:a numeroidaan 0:sta $n-1$:hteen ja näiden jokaisen palvelimet numeroidaan 0:sta n^k-1 :teen. Kerrokset *k* ja *k - 1* yhdistyvät siten että *i*:s palvelin (*i* = $i \in [0, n^l-1]$) *j*:stä $BCube_{k-1}$:sta ($j \in [0, n-1]$) yhdistyy tasolla *k* porttiin *j* tason *k* kytkimessä nro *i*.

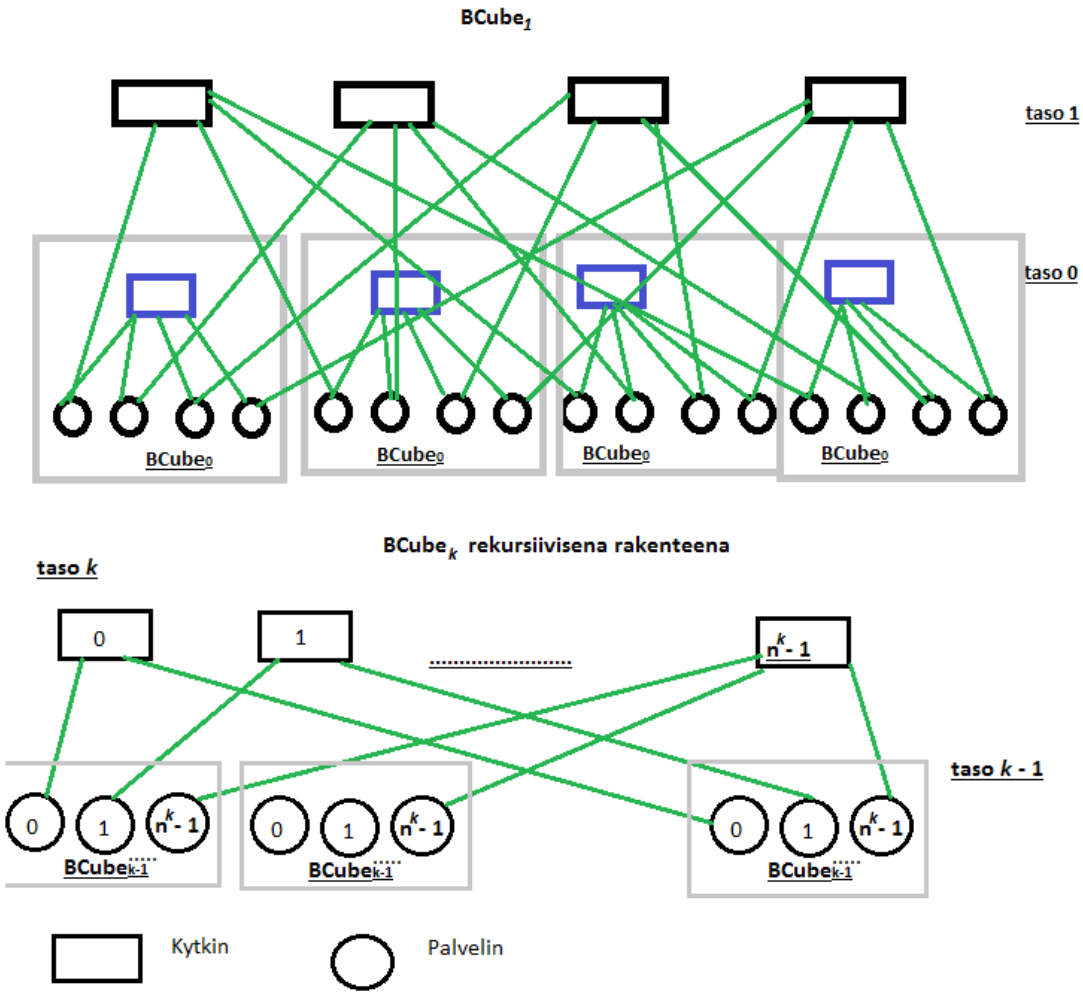
Kuvassa 2.3) esitellään ylempänä $BCube_1$ jossa *n* = 4 ja jokaisella palvelimella on 2 porttia. Alempi kuva taas esittää kuinka $BCube_k$ rakentuu rekursiivisesti *n*:stä $BCube_{k-1}$:sta ja n^k kytkimestä joissa on jokaisessa *n* porttia.

BCubessa kaikki linkit ovat kaksisuuntaisia. Kytkimet yhdistyvät vain palvelimiin eikä kytkin koskaan suoraan yhdisty toiseen kytkimeen. Jokaisen kahden palvelimen välillä on useampi vaihtoehtoinen reitti mikä mahdollistaa MPTCP:n (lisää omassa aliluvussaan) datan kuljetuksessa. $BCube_k$:ssa jokaisen kahden eri palvelimen välillä on k + 1 rinnakkaista polkua.

Zhang [ZXin16] mukaan BCuben suorituskyky heikkenee huomattavasti nopeiden, pienten datavirtojen takia (jotka ovat yksi syy incastille). Tämän vuoksi BCuben kaltaisiin verkkoihin tarvitaan erityisiä datavirtojen järjestelijöitä (flow scheduler).

2.4 Yhteenveto datakeskuksista

Datakeskuksen topologinen rakenne on yleisesti kaksi- tai kolmikerrroksinen jossa palvelinkoneet kytkeytyvät alimman kerroksen kytkimiin ja ylimmän tason kytkimet kytkeytyvät ulos datakeskuksesta eli Internetiin tai toisiin datakeskuksiin. Topologiassa



BCube [GLL09].

reitityksen hoitavat joko kytkimet tai palvelimet. Skaalautuvuuden kannalta topologia tulisi pystyä toteuttamaan halvalla laitteistolla, järjestelmän kuitenkin pystyessä palautumaan laitteistovioista tarvittaessa.

3 Ruuhkan Syyt ja Ilmenemis-muodot

Tässä luvussa esitellään datakeskusten tietoliikenteen ominaisuuksia ja syitä siihen miksi tietoliikenne ruuhkautuu datakeskuksissa. Ensimmäisessä aliluvussa esitellään miten ruuhka ilmenee Internetin kaltaisissa verkoissa, toisin sanoen muualla kuin datakeskusten kaltaisissa ympäristöissä. Tämän jälkeen pohditaan datakeskusten ruuhkautumista jonka jälkeen esitellään eri ruuhkan tyyppejä.

3.1 Tietoliikenteen ruuhka Internetin kaltaisissa verkoissa

Ruuhkan sortumiseksi (Congestion collapse) kutsutaan Internetin yleistä vaivaa jossa tietoliikenteen ruuhka heikentää kommunikaatiota kun paketit syövät jatkuvasti kaistaa mutta eivät pääse koskaan perille määränpäähänsä. Kun termi määriteltiin 1980-luvulla, se tarkoitti tilannetta jossa verkko pysyy vakaassa ruuhkautuneessa tilassa. Tuohon aikaan syynä olivat datan lähettäjien uudelleenlähetyksen ajastimien heikko käyttö jonka vuoksi myöhästyneitä paketteja lähetettiin turhaan uudestaan. Myöhemmät TCP:n implementaatiot korjasivat tämän ongelman. Ongelma on kuitenkin yhä läsnä ja sen välttämiseen tarvitaan ruuhkanhallintaa. Osassa alan kirjallisuutta ruuhkanhallinta ja datavirtojen aikataulutus (flow scheduling) käsitetään samaksi, osassa taas ne erotetaan toisistaan.

Toinen yleinen ongelma Internetin tietoliikenteessä on kaistan epäreilu jakautuminen kilpailevien datavirtojen kesken. Tähän on monia syitä, esimerkiksi sovellukset jotka eivä reagoi kunnolla ruuhkaan ja lisäksi Internet-protokollat itsessäänkin voivat aiheuttaa resurssien epäatasaista jakautumista. Streamatulla medialla esimerkiksi on todettu olevan haitallisia vaikutuksia perinteisempään tietoliikenteeseen Internetissä; streamattu media ei vastaa hyvin verkon ruuhkaan ja voi näin pahentaa ruuhkan sortumista ja epäreilua kaistan jakamista [AVS04].

Internet Congestion Control Research Group (ICCRG) on listannut haasteita liittyen Internetin ruuhkanhallintaan [PWS11]. Tässä mainitaan muun muassa verkon korruptiosta johtuva datan häviäminen (johon ruuhkanhallinta-algoritmit reagoivat väärällä tavalla koska ne luulevat datahäviön johtuvan ruuhkasta), se että TCP ei ruuhkanhallinnassaan reagoi pakettien kokoon vaan kaikkia paketteja kohdellaan samalla lailla, ruuhkanhallintamekanismien soveltaminen monien domainien kesken johon liittyvät mm. turvallisuuteen liittyvät seikat (Internet on jakautunut itsenäisiin järjestelmiin jotka ovat keskenään heterogeenisiä) ja datan lähettäjät ja vastaanottajat jotka toimivat toisiaan vastaan (esimerkiksi ei ole aina lähettäjän edun mukaista vähentää ruuhkaikkunansa kokoa jos lähettäjä olettaa muiden lähettäjien vähentävän oman ruuhkaikkunansa kokoa).

3.2 Datakeskusten ruuhka

Datakeskuksissa on runsaasti yhdestä moneen (many-to-one) ja monesta moneen (many-to-many) liikennettä. Monilla ohjelmilla joiden laskenta suoritetaan datakeskuksissa on asetettu edellytykset aikarajojen suhteen. Tällöin palvelimien pitäisi tulla kyetä suorittamaan haluttu toiminto käyttäjälle kymmenien millisekuntien sisällä, muuten käyttäjäystävällisyys luonnollisesti kärsii. Tällaisista ohjelmista syntyvien pienien datavirtojen tulisi päästä kulkemaan suhteellisen vapaasti verkossa ja näin ollen linkkien tulisi olla vain vähän puskuroitu kullakin ajan hetkellä. Matala kytkinten puskurien kuormitus (low link buffer occupancy) on datakeskuksissa usein vastakkain suurten datavirtojen edellyttämää korkeaa läpisyöttöä koska jos kytkinten puskurit ovat vain vähän täynnä, suuremmat datavirrat eivät pääse etenemään tehokkaasti.

Kirjallisuudessa datakeskuksille tyypilliset datavirrat on lähteestä riippuen jaettu kahteen tai kolmeen kategoriaan riippuen siitä kuinka suurista datavirroista puhutaan. Yleensä ne jaetaan matalaa kytkinten puskurien kuormitusta vaativiin pieniin datavirtoihin ja korkeaa läpisyöttöä vaativiin suuriin datavirtoihin, toisinaan näiden väliin keskelle on joissain lähteissä määritelty kolmas kategoria. Alizadeh et al. tutkimuksessa [AJP11] on määritelty datakeskusten datavirtojen koostuvan kolmesta tyypistä joita ovat räjähdysmäiset tietokantahauista johtuvat viestit (bursty query traffic, 2KB -20 KB), aikarajoille herkät lyhyet viestit (delay-sensitive short messages, 50 KB - 1 MB) ja korkeaa läpisyöttöä edellyttävät pitkät viestit (throughput-intensive long flows, 1 MB - 50 MB). Suurin osa on pieniä (10KB). Zhang [ZXin16] mukaan pienien datavirtojen osuus datakeskusten koko liikenteestä (datavirtojen määrän suhteen) on 99 % mutta koko datamäärän osalta pienet datavirrat siirtävät vain 10 %. Tämän lisäksi enemmän kuin puolet ajasta yksittäisellä verkon palvelimella liikkuu keskimäärin 10 datavirtaa samanaikaisesti. Sisäisen liikenteen lisäksi datakeskuksen pitää myös käsitellä ulkoa tulevaa (esim. toisesta datakeskuksesta tai Internetistä) saapuvaa tietoliikennettä, esimerkiksi jos loppukäyttäjä käyttää pilvipalvelua.

Datakeskusten liikennettä tutkitaan yleensä keräämällä SNMP (Simple Network Management Protocol)-dataa datakeskuksista ja analysoimalla tietoliikenteen hetkellisiä ja spatiaalisia kaavoja sekä tarkkailemalla millä tahdilla paketteja häviää kytkimissä. On todettu että suurin osa datakeskusten datavirroista on pieniä ja kestävät muutama sata millisekuntia. Pilvipalveluita ajavissa datakeskuksissa suurin osa liikenteestä (75 %)

pysyy räkin sisällä kun taas yliopistojen ja yritysten datakeskuksissa 40-90 % liikenteestä käy räkin ulkopuolella kiertäen koko datakeskuksen. Topologian kannalta katsottuna eniten datan häviämistä näyttää tapahtuvan sisäkerroksessa [ZBC16].

Seuraavissa aliluvuissa esitellään datakeskusten ruuhkan tyyppejä.

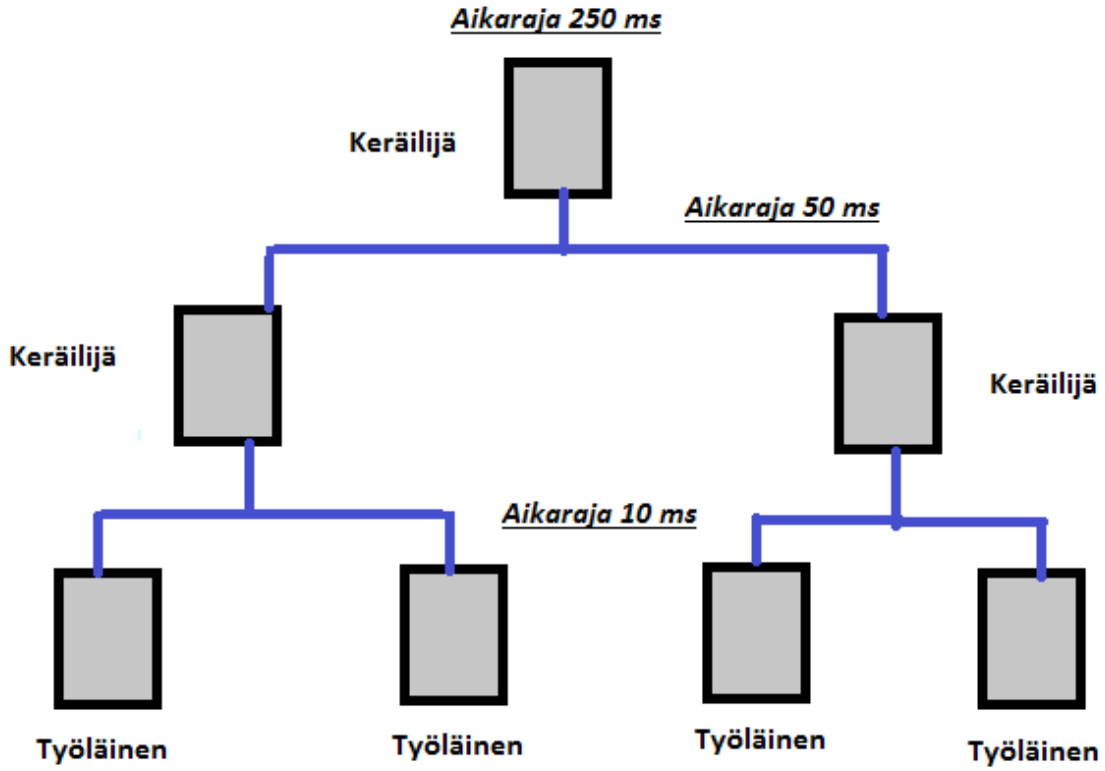
3.3 TCP Incast

Esimerkkitilanne ongelmasta on ohjelmisto kuten esimerkiksi Googlen MapReduce jossa toimii kahdenlaisia aliohjelmia, työläisiä (worker) ja keräilijöitä (aggregator): keräilijä jakaa laskennan useammalle työläiselle ja kun työläiset ovat saaneet laskennan suoritettua ne palauttavat laskennan tuloksen vastaavalle keräilijälle joka taas palauttaa tuloksen jollekin ylemmän tason keräilijälle ja niin edelleen. Kun useamman työläisen laskennan tulokset lähetetään samanaikaisesti keräilijälle, keräilijän päässä datapaketteja vastaanottava puskuri tulee helposti täyteen minkä johdosta paketteja pudotetaan lähetyksestä ja aikakatkaisun myötä työläinen joutuu lähettämään paketin uudestaan. Haun indeksointi (search indexing) on vastaava esimerkki missä yhden sanan toistuvuus pitää laskea useassa dokumentissa. Laskenta hajautetaan usealle palvelimelle jotka kaikki ovat vastuussa joistakin dokumenteista paikallisella kovalevyllään ja vasta kun kaikki palvelimet ovat palauttaneet laskurinsa palvelun pyytäjälle, voidaan lopullinen vastaus tuottaa. Tämän tyyppistä liikennettä kutsutaan kirjallisuudessa nimellä partition/aggregate (kuva 3.3.1).

Yleisesti ottaen ohjelmalle jolle on tärkeää pieni latenssi (esim. hakukoneet) tällainen ruuhkautuminen vaikuttaa selkeästi käyttäjäystävällisyyden heikentymiseen. Kun monet synkronoidut palvelimet muodostavat yhteyden samanaikaisesti saman pullonkaulalinkin kautta, näiden yhteyksien sanotaan olevan barrier synchronized-tilassa; lopullinen palvelun suorituskyky riippuu hitaimmasta yhteydestä joka kärsii

aikakatkaisuista (timeout) pakettien häviämisen takia. Tilannetta kutsutaan nimellä TCP Incast. Incastin seurauksena linkin (jossa incast tapahtuu) läpisyöttö voi pudota jopa 90 % alle maksimista.

Datakeskuksessa on tyypilliset ominaisuudet jotka ovat edellytykset incastille; linkkien suuri kaista (high bandwidth links), pienet latenssit tiedonsiirrossa, pienet puskurimuistit räkkien ja palvelimien välisissä kytkimissä, barrier synchronized-ilmiö

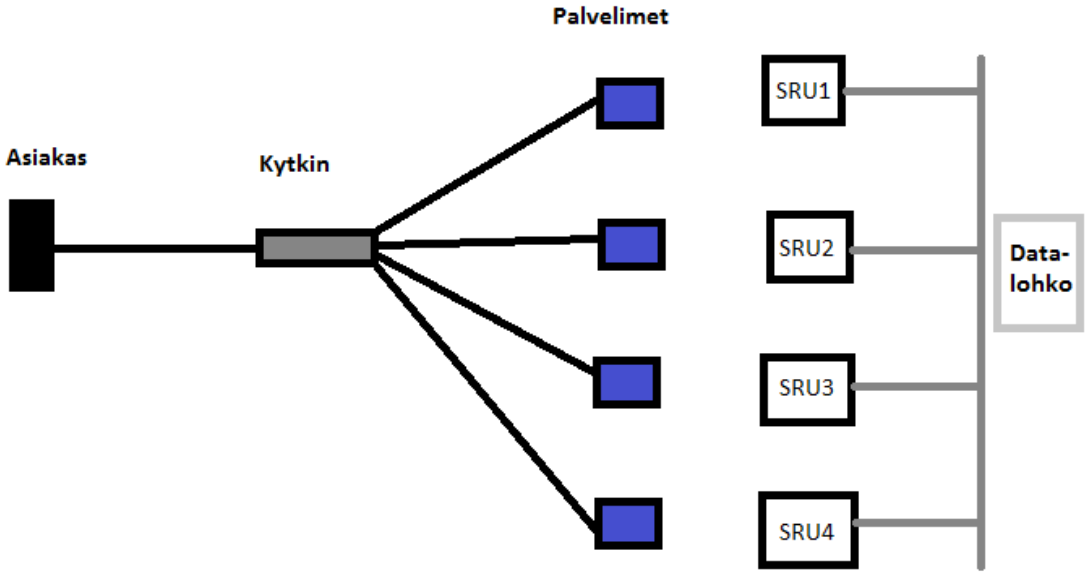


Kuva 3.3.1: Incast tapahtuu usein partition/aggregate-tyyppisen liikenteen vuoksi [AM15].

(asiakaskone pyytää dataa palvelimilta lohkoissa eikä voi pyytää seuraavaa lohkoa ennen kuin edellinen lohko on saapunut) yleisyys (mitä aiheuttavat MapReducen kaltaiset algoritmit) sekä palvelimet jotka toimittavat suhteellisen pieniä datamääriä asiakkaalle kerrallaan [WFG10]. Jos useampi ohjelma lähettää dataa samanaikaisesti yhden kytkimen kautta, tämä lisää incastin todennäköisyyttä.

Incast tapahtuu yleensä räkin palvelimia yhdistävässä kytkimessä (top of rack-switch) johon datan vastaanottaja (jolle useampi palvelin samanaikaisesti lähettää) on yhteydessä (kuva 3.3.2). Incast voi syntyä sekä räkin sisäisessä että ulkopuolisessa liikenteessä; sisäisessä liikenteessä kaikki datan lähettäjät sijaitsevat samassa räkissä ja lähettävät samaan räkkejä yhdistävään kytkimeen jota myös vastaanottaja kuuntelee, ja räkin ulkoupuolisessa liikenteessä lähettäjät sijaitsevat eri puolilla datakeskusta mutta kaikki lähettävät samalle vastaanottajalle.

Tutkijoiden kokeiden perusteella olennaisin syy incastiin ovat TCP:n aikakatkaisut (timeouts). Aikakatkaisu seuraa kun dataa lähetetään nopeasti samaan aikaan monelta palvelimelta mikä helposti kuormittaa kytkimen puskurin johtaen datapakettien runsaaseen katoamiseen. Kun yksi tai useampi palvelin odottaa aikakatkaisun tapahtumista, muut jo lähetyksensä loppuun saaneet palvelimet joutuvat odottamaan



Kuva 3.3.2: Incast-skenaario: asiakas pyytää dataa useilta palvelimilta jotka kaikki säilyttävät osaa datalohkosta (SRU=server request unit); kaikki palvelimet lähettävät oman osansa vastauksesta saman kytkimen kautta (josta tulee pullonkaula) [AM15].

palvelimia joiden lähetykset ovat kesken koska yleensä datakeskuksessa seuraavaa datalohkoa ei voida pyytää ennen kuin edellinen on vastaanotettu kokonaan. Nämä aikakatkaisut johtavat satojen millisekuntien latensseihin mutta datakeskuksissa tiedonsiirtoviiveet (round trip delay time) ovat tavallisesti kymmenestä sataan mikrosekuntia. Tämän seurauksena verkon läpisyöttö voi alentua jopa 90% ja on erityisen haitallista ohjelmille joille on tärkeää pieni latenssi. Sreekumari ja Jung [SJ15] listaavat kolme syytä aikakatkaisuille jotka ovat pakettien häviäminen datalohkojen loppupäästä, pakettien häviäminen datalohkojen alkupäästä sekä uudelleen lähetettyjen pakettien häviäminen.

Yrittäessä välttää incastia ongelmana on säätää ruuhkaikkunan tai lähetysnopeuden kokoa. Ikkunan / nopeuden tulisi olla tarpeeksi pieni incastin välttämiseksi mutta samalla tarpeeksi suuri tehokkaan suorituskyvyn aikaansaamiseksi ja tarpeeksi toimiva muissakin skenaarioissa kuin incast.

Incastista on tullut eräs keskeisiä ongelmia datakeskuksissa. Ongelmaan on kehitetty ratkaisuja jotka tähtäävät tiettyjen sovellusten käyttöön [PW11] mutta yleinen ratkaisu kuljetuskerroksen tasolla luonnollisesti helpottaisi ohjelmistokehittäjien työtä.

Zhang ja Ansari listaavat keinoja joilla incastia on yritetty torjua [ZA13]. Palvelinten (datan lähettäjien) määrää ja lähetysnopeutta voidaan rajoittaa mutta nämä ratkaisut ovat yksilöllisiä tietyille datakeskusten asetuksille. Kytkinten puskurien kasvattamisella voidaan siirtää incastin tapahtumista myöhemmäksi mutta kytkimien ja reitittimien kapasiteetin lisääminen on kallista ja suuretkin kytkinten puskurit voidaan nopeasti ylikuormittaa nopeilla linkeillä. Yksittäisen palvelimen tallentamaa datamäärää (server request unit) voidaan kasvattaa mutta yleensä käyttäjät pyytävät pieniä datamääriä kerrallaan.

Tseng et al. [TYP16] mukaan incastin torjuntaan tarkoitetut algoritmit voidaan ryhmitellä kolmeen kategoriaan: algoritmit jotka pyrkivät vähentämään uudelleenlähetyksen ajastimen (retransmission timeout) arvoa, algoritmit jotka pyrkivät reagoimaan ruuhkaan ennaltaehkäisevästi (ennen kytkimen puskurin ylikuormitusta) sekä algoritmit jotka pyrkivät hyödyntämään vaihtoehtoisia polkuja lähettäjän ja vastaanottajan välillä. TCP:ssa uudelleenlähetyksen ajastin on tyypillisesti vähintään 200ms mikä on pitkä aika odottaa aikakatkaisuun reagoimista datakeskuksessa. Ajastimen vähentäminen on luonnollinen ratkaisu incastiin sillä ajastimen väärät konfiguraatiot ovat yksi olennaisia syitä incastiin mutta se ei kuitenkaan estä kytkimen puskurin täyttymistä. Lisäksi matalan ajastimen käyttö edellyttää kehittyneempää laitteistoa.

3.4 TCP Outcast

Incastiin liittyy keskeisesti verkon kaistan liian alhainen käyttö. TCP Outcastissa kyse on siitä että kaista ei jakaudu reilusti eri datavirtojen kesken [PDH12, QSS13].

Datakeskusten käytössä TCP:n läpisyöttö on käänteisesti verrannollinen tiedonsiirtoviiveeseen; toisin sanoen kun eri datavirrat joilla on eri viive jakavat saman pullonkaulalinkin, datavirrat joilla on suurempi viive saavat suuremman osan kaistasta käyttöönsä kuin datavirrat joilla on pienempi viive [PDH12]. Esimerkiksi kun ohjelmistopäivityksen synnyttämä suurempi joukko datavirtoja ja hakuoperaation tuottama pienempi joukko datavirtoja saapuvat kytkimen eri sisääntuloportteihin ja tähtäävät samaan ulostuloporttiin, hakuoperaation tuottamat datavirrat saavat huomattavasti vähemmän kaistaa käyttöönsä kuin ohjelmistopäivityksen tuottamat. Tätä tilannetta jossa suuremmat datavirrat ikään kuin karkottavat pienemmät datavirrat

kutsutaan TCP Outcastiksi.

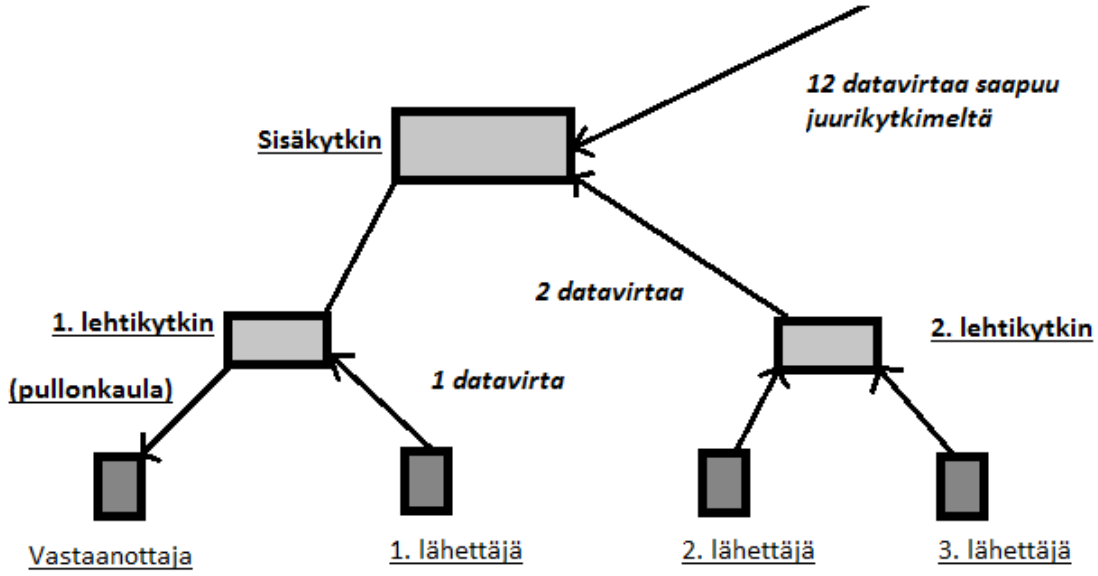
Kuvan 3.4) verkossa on yksi sisäkytkin, kaksi lehtikytkintä (räkkikytkintä), kolme lähettäjää ja yksi vastaanottaja. Datakeskukselle tyypillisesti palvelin joka on pyytänyt datalohkoa voi pyytää seuraavaa datalohkoa muilta palvelimilta vasta kun edellinen datalohko on kokonaan vastaanotettu. Datavirrat joilla on pienempi viive joutuvat odottamaan kunnes datavirrat joilla on suurempi viive ovat lähettäneet osuutensa datalohkosta. Kun paketteja katoaa kytkimen puskurin ollessa täynnä, datavirrat joilla on pienempi viive menettävät enemmän paketteja kuin datavirrat joilla on suurempi viive ja tämän vuoksi datavirtojen joilla on pienempi viive läpisyöttö kärsii enemmän.

Datakeskusten rakenteen vuoksi outcast tapahtuu pääasiallisesti kahdesta syystä: ensinnäkin kytkimet ovat tavallisesti markkinoiden halvempaa osastoa jotka käyttävät pakettien puskuroinnissa taildrop-sääntöä eli jos puskuri on täynnä, paketit pudotetaan. Taildrop-säännön aikaansaamaa tilannetta jossa linkin kaista jakautuu epäreilusti useamman datavirran kesken kutsutaan nimellä *port blackout*; toisen datavirran onnistuu lähettää portista läpi ja toisen paketteja häviää koska ulostuloportin puskuri on täynnä.

Toiseksi, koska MapReducen kaltaiset ohjelmistot ovat datakeskuksissa yleisiä ja niitä suorittavia solmuja voi periaatteessa olla missä tahansa päin verkkoa, kytkimien eri sisääntuloporteille saapuu helposti eri suuruisia datamääriä samanaikaisesti. Lähettäjät sijaitsevat eri etäisyyksillä vastaanottajasta mikä saa aikaan sen että pullonkaulalinkin eri sisääntuloporteille saapuu eri kokoisia datavirtoja jotka kilpailevat samanaikaisesti samasta ulostulo-portista.

Prakashin ja kumppaneiden [PDH12] mukaan ratkaisu outcastiin on varmistaa että ulostuloportista kilpailevat datavirrat kärsivät yhtä paljon pakettien pudottamisesta. He esittävät eri ratkaisutapoja ongelmalle.

RED (random early detection) on vaihtoehto taildrop-säännölle: paketit pudotetaan (tai merkitään esim. ECN:ää käyttäessä) riippuen ennalta määritellystä todennäköisyydestä. Mitä enemmän täynnä kytkimen puskuri on, sitä suuremmalla todennäköisyydellä



Kuva 3.4: TCP Outcast [QSS13].

paketit pudotetaan / merkitään. RED ei kuitenkaan jaa läpisyöttöä datakeskuksissa tasaisesti.

SFQ (stochastic fair queuing) on menetelmä aikatauluttaa datavirrat tavoitteena että jokainen datavirta saa yhtä reilusti kaistaa käytttöönsä riippumatta datavirran viiveestä. Ideana on jakaa kytkimestä ulospäin menevä data lokeroihin (lokeroiden määrä on ennalta määriteltävä luku) siten että samaan lokeroon kuuluvat datavirrat saavat saman verran kaistaa.

Koska ainakin oletusarvoisesti palvelimien sijaintia verkossa ei voi muuttaa, eräs ratkaisu on varmistaa että kaikki datavirrat siirtyvät samanlaisia verkon polkuja pitkin ja näin ollen datapaketit jakautuvat tasaisesti eri linkkien ja sisääntuloporttien kesken. Equal-length Routing-nimellä tunnettu strategia voidaan toteuttaa reitittämällä jokainen datapaketti joko satunnaisesti tai deterministisesti verkon juurikytkimeen. Satunnaisessa ratkaisussa datapaketti reititetään satunnaisesti valittuun juurikytkimeen kun taas deterministisessä ratkaisutavassa reitti kohteena olevaan juurikytkimeen määritellään kohdeosoitteen perusteella. Equal Length Routingin ansiosta, sekä satunnaisen että deterministisen ratkaisutavan avulla, port blackout vältetään ja kaikki kilpailevat datavirrat kärsivät tasaisesti datapakettien menettämisestä; lisäksi tiedonsiirtoviive on sama kaikille datavirroille ja niiden ruuhkaikkunat kasvavat samaan tahtiin ja näin ollen pullonkaulalinkin kaista jakautuu reilusti eri datavirtojen kesken. TCP Pacing on myös esitetty ratkaisuksi outcastille. Näiden eri tyypppisten ratkaisujen sopivuus outcastin

torjumiseen riippuu kuitenkin datakeskuksen vaatimuksista, tietoliikenteen luonteesta sekä topologiasta. Topologiat joissa juurisolmuja on useampi (esim. FatTree) epäreilu kaistan jakautuminen on yleistä ja näin lisää outcastin todennäköisyyttä.

3.5 Latenssi

Koska datakeskuksessa aliohjelma on yleensä samaan aikaan työläinen ja keräilijä, se käsittelee samaan aikaan pieniä viestejä (tietokantakyselyt) ja suuria viestejä (esim. päivitykset). Kun kytkimeen osuu samaan aikaan isoja ja pieniä datavirtoja, niin vaikka pakettien pudottamista ei tapahtuisi niin pienemmät datavirrat joutuvat odottamaan isompien datavirtojen perille pääsyä. Lähettäjän pitää pienentää ruuhkaikkunansa kokoa ja tarvitsee näin ollen enemmän aikaa saadakseen lähetettyä datan kokonaan ja taas tämän seurauksena uudelleenlähetyksen ajastimen tulee olla suurempi kuin suurin mahdollinen verkon tiedonsiirtoviive (RTT) ja tämä johtaa siihen että datavirran lähetys ei onnistu sille asetettuun aikarajaan mennessä. Tämä sen kanssa että datakeskuksissa lyhyitä datavirtoja esiintyy usein räjähdysmäisesti johtaa siihen että lyhyiden datavirtojen paketteja häviää usein [SJ15].

3.6 Viivytetyt ACKit

Viivytetty ACK tähtää TCP:n ACK-kuittausten vähentämiseen jotta takautuva liikenne vastaanottajasta lähettäjään olisi pienempi. Viivytetty ACK tarkoittaa useampien ACK-pakettien yhdistämistä samaksi ACK-signaaliksi. Datakeskuksissa tiedonsiirtoviiveet ovat tyypillisesti pienä minkä vuoksi viivytetyn ACKin käyttö johtaa jatkuviin aikakatkaisuihin koska lähettäjä joutuu kauemmin odottamaan [SJ15].

3.7 Yhden polun TCP

Suurin osa datakeskusten ohjelmistoista käyttävät yksipolkuista TCP:ta (single path TCP), toisin sanoen data kulkee yhtä polkua pitkin kohteeseen. Tämän vuoksi polku ruuhkautuu helposti ja seurauksena läpisyöttö heikkenee ja kaista jakautuu epätasaisesti datavirtojen kesken. MPTCP on tätä varten kehitetty / kehitteillä oleva standardi mikä mahdollistaa liikenteen siirtämisen ruuhkautuneilta poluilta vähemmän ruuhkautuneille. Tämä tietysti edellyttää että topologia sallii vaihtoehtoisten polkujen käyttämisen kahden palvelimen välillä. MPTCP:sta puhutaan lisää myöhemmin omassa aliluvussaan.

3.8 Yhteenveto datakeskusten ruuhkasta

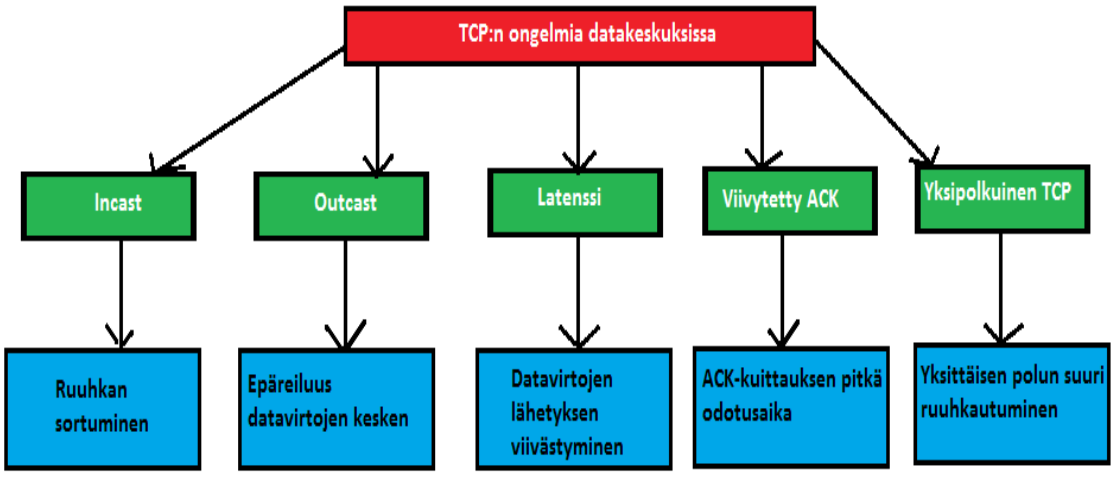
Datakeskus ruuhkautuu helposti johtuen tietoliikenteen luonteesta lisättynä siihen minkälaisia ohjelmistoja datakeskuksissa ajetaan. Datakeskuksissa on lyhyet tiedonsiirtoviiveet sekä paljon kaistaa käytössä mutta tämä yhdistettynä hajautettua laskentaa suorittavien ohjelmistojen generoimaan dataan aiheuttaa hankalien pullonkaulojen syntymistä ympäri verkkoa.

Incast seuraa käyttäjän pyytäessä datalohkoa jonka osia säilytetään eri palvelimilla jolloin palvelimet vastaavat samanaikaisesti mikä ylikuormittaa lähettäjien ja vastaanottajan välillä olevan kytkimen puskurin, johtuen toistuviin datapakettien pudotukseen, aikakatkaisuihin ja pakettien uudellenlähetykseen heikentäen läpisyöttöä.

Outcast syntyy erikokoisten datamäärien kilpaillessa samasta linkin ulostuloportista jossa suuret datavirrat saavat etulyöntiaseman pienempiin datavirtoihin pienien datavirtojen joutuessa odottamaan suurempien datavirtojen lähetystä loppuun.

4 TCP:n ruuhkanhallinta

Datakeskusten kuljetusprotokollat ovat pääosin TCP (transmission control protocol) tai UDP (user datagram protocol), suurimman osan tietoliikenteestä ollen kuitenkin TCP:ta. Po et al. [PJP16] mukaan TCP kuljettaa 85% Internetin ja pilvidatakeskusten liikenteestä. Alunperin TCP on suunniteltu WAN (wide area network)-käyttöön jolle on ominaista suhteellisen suuret latenssit ja matala kaista. TCP on Internetissä



Kuva 3.8: TCP:hen liittyviä ongelmia datakeskuksissa [SJ15].

kuljetuskerroksessa pääasiallisesti käytetty protokolla ja siitä on lukuisia variaatioita. Internetissä ruuhkanhallintaa suorittavat TCP:n ruuhkaikkunaa säätelevät algoritmit joita ajavat verkon isäntäkoneet, sekä AQM (active queue management)-algoritmit joita suorittavat kytkimet ja reitittimet. Ruuhkanhallinta pyrkii varmistamaan tehokkaan linkkien hyödyntämisen, pienet datapakettien jonotusajat ja resurssien reilun jakamisen käyttäjien kesken.

TCP:n ominaisuuksia on että se käyttää kaiken saatavilla olevan kaistan (tai ainakin yrittää jos vapaata kaistaa löytyy) ja että se on reilu protokolla; jos useat datavirrat siirtyvät samaa linkkiä pitkin, jokainen datavirta saa käyttöönsä yhtä paljon kaistaa. TCP:lle on vuosien saatossa kehitetty lukuisia eri variantteja jotka voidaan jakaa eri kategorioihin. Näitä ovat häviöön perustuvat algoritmit (esim. NewReno, Westwood+, HighSpeed, BIC) joissa datapakettien häviäminen on merkki ruuhkasta, viiveeseen perustuvat algoritmit (mm. Vegas) jossa ruuhkaantuminen päätellään siitä että pakettien jonotusaika pitenee kytkimen puskurin täyttymisen myötä, sekä algoritmit jotka ovat näiden kahden edellisen ryhmän hybrideitä (esim. YeAH).

4.1 TCP-algoritmeja

Seuraavaksi esitellään joitain TCP-algoritmeja jotka ovat käytössä Internetissä. Jotkut ovat sitä mieltä että datakeskuksille ei tulisi suunnitella erillisiä algoritmeja yksinomaan datakeskusten käyttöön vaan Internetissä käytettyjen TCP-algoritmien tulisi pystyä suoriutumaan myös datakeskuksissa [NGS16, LJM12]). Seuraavaksi esitellään TCP:n muunnoksista New Reno, Vegas sekä YeAH. New Reno edustaa pakettien häviämiseen perustuvia algoritmeja (loss-based), Vegas edustaa pakettien viiveeseen perustuvia algoritmeja (delay-based) ja YeAH edustaa hybridi-algoritmeja jotka ovat kahden edellisen ryhmän risteytyksiä.

TCP:ssä ruuhkaikkuna (congestion window) määrittää kuinka paljon datan lähettäjä saa siirtää dataa kerralla ennen kuin lähettäjä on saanut kuittauksen datan perille saapumisesta vastaanottajan generoiman ACKin (acknowledgement packet) kautta jonka vastaanottaja lähettää lähettäjälle kun data on vastaanotettu. Vastaanottajaikkuna (receive window) taas kertoo kuinka paljon vastaanottaja voi vastaanottaa dataa ennen ACKin lähettämistä lähettäjälle. Korkealla tasolla TCP:n ruuhkanhallinta voidaan jakaa kahteen osaan. Hitaan aloituksen aikana ruuhkaikkunaa kasvatetaan aina kun lähettäjä

on saanut ACKin vastaanottajalta kun datapaketti (tai ennaltamäärätty määrä *x* datapaketteja jolloin kyseessä on ns. kumulatiivinen ACK) on saapunut perille. Tämän seurauksena lähettäjän ruuhkaikkunan koko kasvaa hitaan aloituksen aikana eksponentiaalisesti; tämä jatkuu siihen asti kunnes joko ruuhkaikkunan koko on ylittänyt hitaan aloituksen kynnysarvon (ssthresh), aikaisemmin lähetetyn datapaketin ACK ei ole saapunut lähettäjälle tietyn ajan sisällä tai ruuhkaikkuna on suurempi tai yhtä suuri kuin vastaanottajaikkunan maksimikoko. ACKin saapumatta jääminen on yleensä merkki siitä että paketti, josta lähettäjä odotti ACKia, ei ole saapunut vastaanottajalle ja on todennäköisesti hukkunut matkalla. Hitaan aloituksen jälkeen lähettäjä siirtyy ruuhkanvälttelyyn (congestion avoidance phase) jonka aikana ruuhkaikkunaa kasvatetaan lineaarisesti.

4.1.1 New Reno

New Reno noudattaa standardi AIMD-ruuhkanhallintaa (Additive Increase, Multiplicative Decrease) ja on muunnos TCP Renosta. Hitaan aloituksen aikana lähettäjän saadessa ACKin ruuhkaikkunaa kasvatetaan seuraavasti: $cwnd = cwnd + 1$ eli ruuhkaikkuna kasvaa eksponentiaalisesti. Hitaan aloituksen aikana kun *cwnd* on saavuttanut hitaan aloituksen kynnysarvon *sstresh,* lähettäjä hidastaa ruuhkaikkunan kasvattamista; *sstresh*:n saavuttamisen jälkeen *cwnd* kasvaa yhdellä jokaisen RTT:n (round trip time) välein tai $\frac{1}{cwnd}$ verran jokaisen ACK-kuittauksen jälkeen: $cwnd = cwnd + \frac{1}{cwnd}$. RTT kertoo kuinka kauan kuluu signaalin lähettämisestä vastaanottajalle siihen asti kunnes lähettäjä on saanut vastaanottajalta signaalin kuittauksen.

Kolmen duplikaatti-ACKin saapuminen lähettäjälle kertoo hävinneestä datasta ja tällöin lähettäjä aloittaa nopean uudelleenlähetyksen; lähettäjä uudelleenlähettää kadotetun datasegmentin, puolittaa hitaan aloituksen kynnyksen ja muokkaa ruuhkaikkunaa:

$$sshthresh = \frac{cwnd}{2}$$

$$cwnd = ssthresh + 3 \quad .$$

Nopea uudelleenlähetys jatkuu siihen asti kunnes kadonneesta segmentista on saatu

ACK.

New Reno modifioi TCP Renon nopeaa uudelleenlähettämistä siten että New Reno reagoi datan osittaisiin kuittauksiin; toisin sanoen New Reno reagoi niihin ACK-paketteihin jotka kuittaavat vain osan lähetetystä datasta ennen kuin datan häviäminen on havaittu. New Reno jatkaa nopeaa uudelleenlähetystä yrittäen saada takaisin hävinneitä datasegmenttejä kun nämä osittaiset ACKit ovat saapuneet. Tämä mahdollistaa useampien hävinneiden pakettien uudelleenlähetyksen yhden ruuhkaikkunan aikana silloin kun TCP:n SACK-optio (Selective Acknowledgement) ei ole käytettävissä (SACK:in avulla vastaanottaja ilmaisee lähettäjälle tarkalleen juuri mitkä paketit ovat saapuneet perille ja mitkä ei).

4.1.2 Vegas

TCP Vegas lisää (tai vähentää) lähetysnopeutta varmistaakseen että vastaanottajalle johtavassa linkissä olisi aina mahdollisimman vähän ylimääräisiä paketteja jonossa (muuttuja *backlog* alempana). Ruuhkaikkunan säätely perustuu verkon RTT-mittauksiin ja yhteyden todelliseen läpisyöttöön (toisin sanoen, ei arvioituun läpisyöttöön) koska näiden muutokset kuvaavat verkon dynamiikkaa varsinkin ruuhkan aikana. Vegas laskee varsinaisen läpisyötön (muuttuja *a*) sekä odotetun läpisyötön (muuttuja *e*) ja näiden erotuksen (*backlog*) seuraavasti:

$$a = \frac{cwnd}{RTT}$$

$$e = \frac{cwnd}{BaseRTT}$$

$$backlog = e - a \quad .$$

BaseRTT on pienin yhteyden elinaikana havaituista RTT-mittauksista.

4.1.3 YeAH

YeAH yhdistää eri ruuhkanhallinta-algoritmien suunnittelukriteereitä. YeAH voi olla kahdessa tilassa; nopeassa tilassa ruuhkaikkuna kasvaa aggressiivisemmin ja hitaassa tilassa ruuhkaa pienennetään ennaltaehkäisevästi (Vegasin tavoin). YeAH pystyy päättelemään jos sen kanssa kilpailee Reno-datavirtoja ja säätelee lähetysnopeuttaan varmistaakseen että verkon resursseista kilpaillaan reilusti. Vegas laskee backlogin (kuinka paljon ylimääräisiä paketteja jonottaa kytkimellä) ja ruuhkan määrän

seuraavasti:

$$Q = \frac{(RTT_{min} - RTT_{Base}) \times cwnd}{RTT_{min}}$$ (backlog);

$$L = \frac{RTT_{queue}}{RTT_{Base}}$$ (ruuhkan määrä).

RTT_{Base} on lähettäjän havaitsema minimi RTT (t.s. etenemisviive) ja RTT_{min} on tämän hetkisen ruuhkaikkunan lähetyksen aikana laskettu minimi RTT. Tämän hetkinen arvioitu jonotusviive on $RTT_{min} - RTT_{base}$. Jos $Q < Q_{max}$ ja $L < \frac{1}{d}$, algoritmi on nopeassa tilassa ja muuten hitaassa tilassa. Q_{max} kertoo kuinka paljon paketteja yksittäisen datavirran paketteja voi jonottaa kytkimellä; $\frac{1}{d}$ kertoo kuinka paljon kytkin voi olla enimmillään ruuhkautunut suhteessa BDP:hen (bandwidth delay product). BDP on yhtä kuin linkin maksiminopeus kerrottuna RTT:llä. Q_{max} ja d ovat säädettäviä parametreja joilla algoritmin toimintaan voidaan vaikuttaa.
Muuttujaa Q käytetään hitaan aloituksen kynnysarvon säätämiseksi jos lähettäjä saa kolme duplikaatti-ACKia (joka ilmaisee ruuhkaa) silloin kun YeAH ei kilpaile Reno-datavirtojen kanssa:

$$ssthresh = min[max(\frac{cwnd}{8}, Q), \frac{cwnd}{2}]$$.

Jos taas YeAH kilpailee Renon kanssa, kynnysarvo puolitetaan jotta kilpailu verkon resursseista olisi reilua.

4.2 TCP datakeskuksissa

Nguyen et al. [NGS16] simuloivat eri TCP-algoritmeja 6-FatTree-topologiassa. Tulosten perusteella TCP Vegas, jonka ruuhkanhallinta perustuu latenssiin ja joka pyrkii reagoimaan ruuhkaan jo ennen kuin datapaketteja on kadonnut, pärjää paremmin kuin muut vertailussa olevat datapakettien häviöön perustuvat algoritmit. Kyseiset algoritmit on alunperin suunniteltu ajatellen tiedonsiirtoväyliä joille on ominaista isot latenssit, Vegas sen sijaan tähtää pieneen latenssiin ja datakeskuksissa on yleensä pieni latenssi. Ruuhkaikkunan säätely perustuu odotetun ja todellisen läpisyötön erotukseen. Vegas olettaa että verkon RTT pysyy vakiona siihen asti kunnes maksimi siirtonopeus on saavutettu, toisin sanoen Vegas olettaa että RTT:n kasvu johtuu ainoastaan pakettien

jonottamisesta kytkimen puskurissa. Vegas säätelee lähetysnopeuttaan RTT:n ja yhteyden läpisyötön perusteella varmistaakseen sen että jos verkossa on pullonkaulalinkki, siinä on aina hyvin pieni määrä ylimääräisiä paketteja jonossa (tässä ylimääräinen on yhtä kuin odotetun ja saavutetun läpisyötön erotus). Nguyenin kanssa samoilla linjoilla ovat Lee et al. [LJM12] jotka simuloivat samaan aikaan TCP Vegasia sekä erityisesti datakeskusten käyttöön suunniteltua DCTCP:tä (lisää myöhemmässä luvussa 'DCTCP') dumbbell-topologiassa. Lee et al. tutkivat miten datavirtojen vähittäinen lisääminen vaikuttaa kytkimen puskurissa jonottavien pakettien määrään kun ruuhkanhallinnan suorittaa Vegas tai DCTCP. Tulosten perusteella Vegas rajoittaa kytkimen puskurin pakettijonon pituutta yhtä hyvin kuin DCTCP ja lisäksi Vegasin kohdalla jonon pituudessa on paljon vähemmän vaihtelua kuin DCTCP:n kohdalla. Kuitenkin Vegasin oletus siitä että datakeskuksessa RTT on vakaa ennen kuin verkon maksimi siirtonopeus on saavutettu on kyseenalaistettu [WFG10].

4.3 ECN

TCP:n ruuhkanhallinta-algoritmit voidaan karkeasti ottaen jakaa kahteen ryhmään. Toiset perustuvat RTT-mittauksiin siten että RTT:n kasvu voidaan tulkita ruuhkautumisena. Toisen ryhmän protokollissa kytkimet lähettävät signaalin lähettäjälle ruuhkatilanteen tapahtuessa. ECN (Explicit Congestion Notification) on valinnainen laajennus IP:hen ja TCP:hen. IP-otsakkeessa on ECN:lle 2 bittiä eli 4 codepointia; datan lähettäjä asettaa kolme näistä codepointeista määrittääkseen tukeeko datavirta ECN:ää ja (ECN:ää tukeva) kytkin asettaa neljännen codepointin ilmaistakseen onko ruuhkaa vai ei.

Kun verkon kytkimet tukevat ECN:ää, jos kytkin havaitsee datapaketin saapuessa ruuhkaa, kytkin merkitsee datapaketin otsakkeen CE (congestion experienced)-kentällä (sen sijaan että hylkäisi paketin). Datan vastaanottaja näkee CE-merkityn paketin ja koodaa ECE-signaalin (ECN Echo) ACK-kuittaukseen joka kertoo lähettäjälle ruuhkasta. Lähettäjä vastaanottaa ACKin kautta ECE-signaalin ja tämän seurauksena lähettäjä puolittaa vastaanottoikkunansa ja ja merkitsee seuraavan datalähetyksensä CWR-bitillä (Congestion Window Reduced). Vastaanottaja jatkaa ECE-signaalin lähettämistä lähettäjälle kunnes vastaanottaja on saanut lähettäjältä paketin joka on merkitty CWR-bitillä joka kertoo vastaanottajalle että lähettäjä on reagoinut ruuhkaan ja

vastaanottajan ei tarvitse enää lähettää ECE-signaalia.

Yksinkertainen tapa vähentää verkon linkkien kuormitusta on käyttää AQM (Active Queue Management)-algoritmeja kuten REDia (Random Early Detection). RED on monesti ECN:n kanssa käytetty vaihtoehtoinen tapa valita mitkä paketit tulee merkitä tai pudottaa ruuhkan ilmaisemiseksi (ECN:ssä ruuhkan aikana paketit merkitään mutta yleisesti REDissa paketit hylätään). REDissa verkon kytkimessä tarkkaillaan tilastollisesti mikä on keskimääräinen pakettijonon koko. Jos tämän keskimääräisen jonon koko on ennalta määrättyjen ala- ja ylärajan välissä, saapuva datapaketti merkitään tietyllä ennalta määrätyllä todennäköisyydellä. Jos keskimääräinen jonon koko on alle alarajan, kaikki saapuvat paketit hyväksytään (ei merkitä) ja jos keskimääräinen jonon koko on ylärajaa suurempi, kaikki saapuvat paketit hylätään (merkitään).

5 Datakeskusten ruuhkanhallinta-algoritmeja

Datakeskuksia varten on kehitetty ruuhkanhallinta-algoritmeja jotka ovat monesti TCP:n modifikaatioita. Jotkut ratkaisut on kehitetty sovelluskerroksen tasolla [PW11] ja jotkut ratkaisut on suunniteltu kuljetuskerroksen tasolla. Tässä luvussa esitellään datakeskusten ruuhkanhallinta-algoritmeja jotka pyrkivät kukin ratkaisemaan erilaisia ruuhkan ilmentymiä datakeskuksissa. Mitään varsinaisesti yleispätevää tai parasta ratkaisua tai algoritmia ei esitellä. Myöhemmässä luvussa 6 vertaillaan tässä luvussa käsiteltyjä algoritmeja niiden ominaisuuksien ja käyttötarkoituksien suhteen.

5.1 DCTCP

DCTCP (Data Center TCP) [AGM10, AJP11] on suunniteltu samaan aikaan ylläpitämään korkea läpisyöttö sekä linkkien kuormitus matalana. DCTCP:n toiminta koostuu ECN:n käytöstä sekä ruuhkan määrän arvioimisesta.

Heti kun kytkimen puskurin pakettijonon koko on ylittää ennalta määritellyn ylärajan K , kytkin merkitsee paketin ECN:n CE (congestion experienced)-kentällä ja toimittaa edelleenlähettää paketin vastaanottajalle. Vastaanottaja sisällyttää ECE (ECN Echo)-signaalin lähettäjälle lähetetyssä ACKissa. Lähettäjä vähentää ruuhkaikkunansa kokoa riippuen siitä minkä verran ACK-paketteja on merkitty; mitä enemmän paketteja on

merkitty, sitä enemmän lähetysikkunaa pienennetään. Vastaanottaja lähettää jokaisesta merkitystä paketista ACKin lähettäjälle kertoakseen tasan tarkkaan kuinka monta pakettia on mennyt ruuhkan ylärajan *K* yli. Jos kokonaisen ikkunan verran paketteja on merkitty, lähettäjä puolittaa ruuhkaikkunansa.

DCTCP:ssä ruuhkaikkunan uusi koko määritellään funktiolla joka riippuu muuttujasta

$a=(1-g)\times a+g\times F$ joka arvioi kuinka suuri osa paketeista on merkitty; F määrittää kuinka suuri osa edellisen ruuhkaikkunan paketeista oli merkitty ja $0<g<1$ on painotettu muuttuja. Uuden lähetysikkunan koko W asetetaan funktiolla $W=(1-a/2)\times W$.

Lähettäjä päivittää arvion merkityistä paketeista aina kun yksi ikkunallinen dataa on lähetetty (suunnilleen yhden RTT:n välein). Muuttujan *a* avulla arvioidaan todennäköisyys jolla kytkimen puskurissa jonottavien pakettien määrä on suurempi kuin yläraja *K*. Kun *a* on lähellä nollaa, ruuhka on vähäistä ja ja ruuhkaikkunaa leikataan vain vähän ja kun *a* on lähellä ykköstä, ikkuna leikataan melkein puoliksi eli ruuhka on suurta.

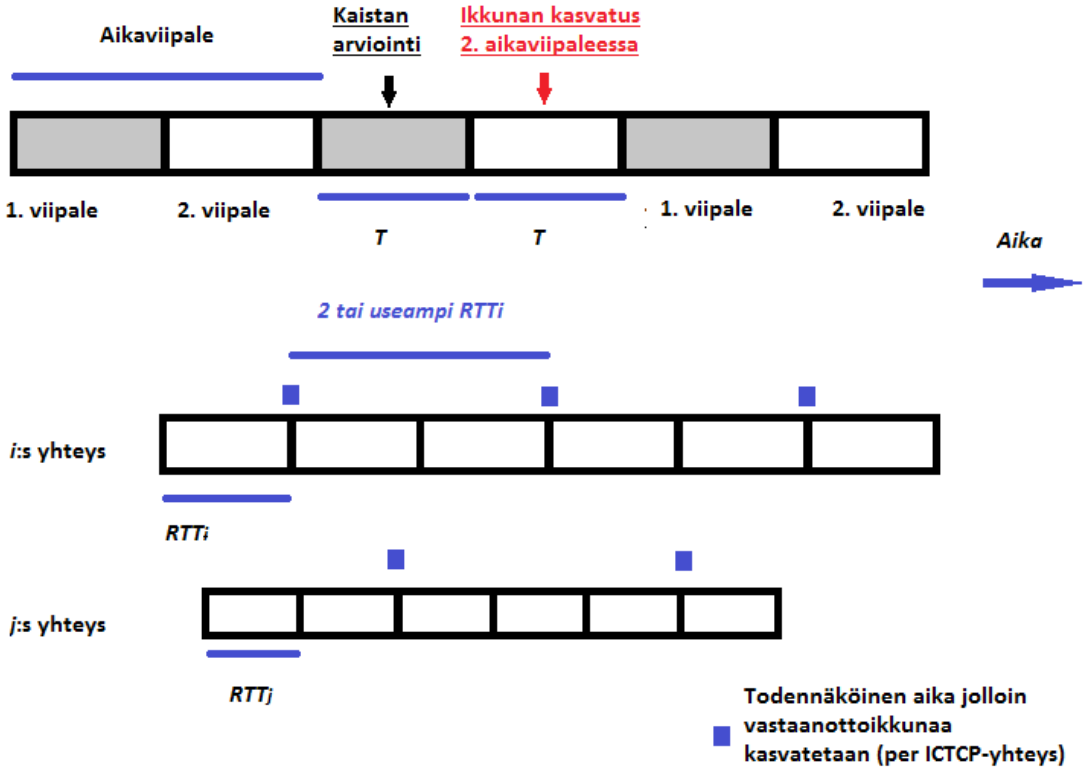
TCP- ja DCTCP-lähettäjän ero on siinä miten ne suhtautuvat kun ne saavat ruuhkasignaalin (eli merkityn ACK-paketin). Muut TCP:n ominaisuudet, kuten hidas aloitus, additiivinen lisäys (additive increase) ja hävinneisiin paketteihin reagoiminen ovat samoja DCTCP:ssä. TCP:ssä lähettäjän vastaanottaessa ruuhkasignaalin ruuhkaikkuna yksinkertaisesti puolitetaan mutta datakeskuksen olosuhteissa tämä johtaa kaistan turhan alhaiseen hyödyntämiseen.

DCTCP:n tehokkuuteen vaikuttaa kaksi muuttujaa; i) kytkimellä käytetty kynnysarvo *K* puskurin jonon koolle; ja ii) muuttuja *a* jota lähettäjä käyttää ruuhkan määrän arvioimiseen.

5.2 ICTCP

Wu et al. [WFG10] esittivät ratkaisun incastiin jossa pelkästään vastaanottaja suorittaa ruuhkanhallintaa (edellä kuvatun DCTCP:n toteutuksessa sekä lähettäjää, vastaanottajaa että näiden välissä olevaa kytkintä joudutaan modifioimaan). Incastissa pullonkaulalinkille lähettää samanaikaisesti palvelinta ja linkin toisessa päässä oleva vastaanottaja säätelee jokaisen saapuvan yhteyden vastaanottoikkunaa. Ratkaisu pitää

ruuhkanhallinta vastaanottajan vastuulla on siitä hyvä että vastaanottaja voi tarkkailla tälle kohdistettua läpisyöttöä ja jäljellä olevaa vapaata kaistaa. ICTCP:ssä (Incast Congestion Control for TCP) vastaanottoikkunaa säädetään vain pienien datavirtojen kohdalla (RTT < 2ms) koska incastia aiheuttavat yleensä vain pienet datavirrat ja näin esimerkiksi datavirrat jotka tulevat Internetistä tai toisesta datakeskuksesta voidaan jättää huomiotta. Vastaanottaja arvioi kuinka suureksi sille osuva läpisyöttö voi nousta päätelläkseen onko kaistaa vapaana tarpeeksi niin paljon että vastaanottoikkunaa voisi kasvattaa. Vastaanottaja ottaa huomioon jokaisen vastaanottajalle saapuvan yhteyden saavutetun ja odotetun läpisyötön sekä saatavilla olevan kaistan linkillä joka on viimeinen lähettäjien ja vastaanottajan välissä. ICTCP käyttää myös erityistä uudelleenlähetyksen ajastinta tehden datan hävitessä uudelleen lähetyksistä nopeampia.
Vastaanottaja tarkkailee sen ja lähettäjien välillä olevalla viimeisellä linkillä saatavilla olevaa kaistaa ylläpitämällä muuttujaa $BW_A = max(0, a \times C - BW_T)$. Tässä C on vastaanottajalle kulkevan linkin kapasiteetti ja BW_T kertoo kuinka paljon kaistaa on käytössä (tällä hetkellä kulutettu) vastaanottajalle johtavalla linkillä. Muuttujalla *a* hienosäädetään funktiota.
Vastaanottoikkunaa suurennetaan ainoastaan siinä tapauksessa jos BW_A on tarpeeksi suuri jonka jälkeen vapaa kaista BW_A kulutetaan. BW_A :n arvioimiseksi ICTCP:ssä aika jaetaan kahteen saman pituisiin aikaviipaleihin (kuva 5.2);
vastaanottaja mittaa ensimmäisessä aikaviipaleessa saapuneen liikenteen ja käyttää tätä laskeakseen saatavilla olevan vapaan kaistan joka on kiintiö sille että vastaanottoikkunaa kasvatetaan. Ensimmäisen aikaviipaleen aikana minkään vastaanottajalle saapuvan yhteyden vastaanottoikkunaa ei kasvateta (mutta saatetaan pienentää). Toisessa aikaviipaleessa mitä tahansa vastaanottoikkunaa saatetaan kasvattaa mutta ainoastaan silloin jos arvioitu summa, joka saataisiin kaikkien yhteyksien läpisyötön yhteenlaskemisesta sen jälkeen kun yksittäistä ikkunaa on kasvatettu, on vähemmän kuin ensimmäisessä aikaviipaleessa laskettu vapaa kaista BW_A . Yhdenkään vastaanottoikkunan pienentäminen ei kasvata BW_A :ta vaan BW_A nollataan vasta seuraavassa aikaviipaleessa joka on taas seuraavassa iteraatiossa ensimmäinen aikaviipale.
Aikaviipaleen kesto on painotettu keskiarvo jokaisen ICTCP-yhteyden RTT:sta



Kuva 5.2: ICTCP:ssä vastaanottaja jakaa ajan T pituisiin aikaviipaleisiin. Ensimmäisessä aikaviipaleessa arvioidaan saatavilla oleva vapaa kaista jonka perusteella toisessa ajanjaksossa vastaanottajaikkunaa kasvatetaan.
Ylempi kuva esittää miten yhteys jaetaan osiin ajan suhteen. Alemmassa kuvassa on esitetty miten joka yhteydelle RTT (Round-trip-time) on yksilöllinen jonka vuoksi jokaisen yhteyden kohdalla kaistan arvio / vastaanottoikkunan säätely kestää tietyn tietyn ajan riippuen yhteydestä [WFG10].

$T = \sum (w_i \times RTT_i)$. Tässä w_i on yhteylle i annettu painoarvo (normalized traffic volume over all traffic). Jokaisen yhteyden i kohdalla vastaanottoikkunaa säädellään yhteydelle i yksilöllisen RTT_i :n perusteella. ICTCP:ssä kontrolli-intervalli (control interval) on (minimissään) 2*RTT sillä yksi RTT kuluu viiveeseen jonka aiheuttaa uudelleen säädetyn ikkunan käyttöönotto ja yksi RTT kuluu läpisyötön arvioimiseen uuden ikkunan perusteella. Jos ICTCP-yhteys on toisessa aikaviipaleessa ja havaitsee että edellisestä vastaanottoikkunan muokkaamisesta on kulunut yli 2 RTT:tä, vastaanottoikkunaa voidaan kasvattaa uusimman havaitun läpisyötön ja vapaan kaistan perusteella.

TCP-Vegasin tavoin ICTCP laskee joka yhteyden kohdalla varsinaisen ja odotetun

läpisyötön mutta käyttää ruuhkaikkunan koon asettamisessa varsinaisen ja odotetun läpisyötön erotuksen suhdetta eikä varsinaista erotusta. Yhteyden *i* odotettu läpisyöttö saadaan kaavalla $e_i = max(m_i, \frac{rwnd_i}{RTT_i})$ missä $rwnd_i$ on yhteyden *i* vastaanottoikkuna ja RTT_i on yhteyden *i* RTT. Varsinainen yhteyden *i* läpisyöttö m_i päivitetään jokaisen RTT_i :n välein. Odotetun ja mitatun läpisyötön erotuksen suhde lasketaan kaavalla $d_i = (e_i - m_i)/e_i$.

Tämän lisäksi ICTCP käyttää kahta kynnysarvoa *y* ja *z* joiden mukaan vastaanottoikkunaa muokataan;

1) Jos $d_i \leqslant y$ tai $d_i \leqslant (MSS_i / rwnd_i)$;

vastaanottoikkunaa kasvatetaan jos yhteys on ajan suhteen toisessa aikaviipaleessa ja vapaata kaistaa on tarpeeksi (MSS_i = maksimi segmentin koko yhteydellä *i*);

2) Jos $d_i > z$;

ja tämä ehto on kestänyt kolmen RTT:n verran, vastaanottoikkunaa pienennetään yhdellä MSS:llä (vastaanottoikkunan minimi koko on 2*MSS).

3) Muussa tapauksessa vastaanottoikkunaa ei muokata.

ICTCP:ssä vastaanottajalla on vaiheina (kuten TCP:ssä lähettäjällä) hidas aloitus ja ruuhkanvälttely. Hitaassa aloituksessa, jos ehto 2 tai 3 on voimassa, uusi yhteys voidaan avata ja ruuhkanvälttelyyn siirrytään jos ehto 1 on voimassa mutta kaistaa ei ole tarpeeksi. Jos hitaan aloituksen aikana kaistaa on tarpeeksi, vastaanottoikkunan koko kaksinkertaistetaan ja ruuhkanvälttelyn aikana vastaanottoikkuna kasvaa maksimissaan yhden MSS:n verran.

Koska datakeskuksissa tavallisesti yhtä palvelua suorittaa samanaikaisesti useampi TCP-yhteys, ICTCP pyrkii siihen että eri yhteydet saavat reilusti kaistaa käyttöönsä. Tämän ICTCP tekee pienentämällä valittujen yhteyksien vastaanottoikkunan kokoa kun yhteenlaskettu vapaa kaista alittaa tietyn kynnyksen (< 0.2 *C*; *C* = linkin kapasiteetti). Yksi MSS leikataan vastaanottoikkunasta niiltä yhteyksiltä joiden vastaanottoikkuna on suurempi kuin kaikkien yhteyksien keskimääräinen yhteenlaskettu ikkunan koko.

5.3 ECN*

Wu et al. [WJL12] sanovat että yksinkertainen kytkimen ECN-logiikan muokkaaminen vähentää incastia sekä pienien ja suurien datavirtojen kilpailua. Esitetty ratkaisu on

nimeltään ECN* jossa verkon kytkimet käyttävät modifioitua ECN:ää ja lähettäjän ja vastaanottajan toimintalogiikkaan ei puututa. Modifioitu ECN tässä tapauksessa tarkoittaa että aina kun kytkin prosessoi sille saapuvan paketin, kytkin vertaa varsinaista jonon pituutta (instant queue length) ennalta määrättyyn kynnysarvoon, toisin kuin standardi-ECN joka vertaa keskimääräistä jonon pituutta kynnysarvoon.

Pakettijonon varsinainen pituus edustaa kaikkien TCP-yhteyksien, jotka ovat samalla pullonkaulalinkillä, ruuhkaikkunaa. Jos jonon pituus on suurempi kuin ennalta määrätty kynnys *h*, saapunut paketti merkitään. Muuttuja *h*:n muokkaaminen on ainoa tapa hallita ECN*:ää.

Wu et al. suorittamien evaluaatioiden perusteella kynnysarvon tulisi olla yhtä kuin verkon BDP eli linkin maksiminopeuden ja RTT:n tulo ja kytkimen puskurin koon tulisi olla kolme kertaa kynnysarvoa suurempi. Kyseinen vaatimus puskurin kokoon nähden on Internetiä ajatellen suuri mutta datakeskuksiin soveltuva.

Simulaatioiden perusteella ECN* ja DCTCP pärjäävät tasavertaisesti tilanteissa joissa verkossa liikkuu vain suuria datavirtoja ja molemmat pärjäävät paljon paremmin tilanteessa jossa muutama lähettäjä yrittää toimittaa pieniä datamääriä suurempien datavirtojen kilpaillessa taustalla kaistasta ja incastissa jossa useampi datavirta kilpailee samasta portista lyhyen ajan sisällä (incast voi tapahtua riippumatta siitä liikkuuko taustalla suuria datavirtoja).

ECN* voi olla laitteistovaatimuksia ajatellen parempi vaihtoa esimerkiksi DCTCP:hen verrattuna koska DCTCP:n käyttö edellyttää TCP-pinon muokkaamista palvelimissa. ECN* vaatii ainoastaan kytkimien modifikaatiota. Wu et al. mukaan ECN*:n logiikka pakettien merkitsemiselle ECN-biteillä ei ole riittävä vaan se tarvitsee tuekseen dequeue marking-mekanismin. Tämä johtuu siitä että ECN*:ssä tieto ruuhkasta ei etene tarpeeksi nopeasti; kun kytkimellä puskuroitujen pakettien määrä ylittää kynnyksen ja saapuva paketti merkitään, kestää jonkin aikaa ennen kuin paketti pääsee jonosta ulos ja näin ollen menee aikaa ennen kuin paketti saapuu vastaanottajalle kertomaan ruuhkasta. Tämän vuoksi ECN* käyttää dequeue markingia; sen sijaan että kytkimelle saapuva paketti merkittäisiin silloin kun se saapuu kytkimelle (ennen kuin se siirretään kytkimen puskuriin jonottamaan), paketti merkitään vasta kun se otetaan pois jonosta.

Simulaatioiden perusteella dequeue marking mahdollistaa useamman incast-lähettäjän

läsnäolon sekä DCTCP:ssä että ECN*:ssä. Simulaatiot myös näyttävät että ECN* pystyy likimain yhtä tehokkaaseen suoritukseen incastissa tarkasteltaessa kolmen tyyppistä ruuhkaskenaariota.

5.4 D²TCP

DCTCP ei ota huomioon ohjelmistoille asetettuja aikarajoja. D²TCP (Deadline-aware Datacenter TCP) on DCTCP:n variaatio jossa datavirroilla joilla on suurempi aikaraja on suurempi prioriteetti niihin datavirtoihin nähden joiden aikaraja ei ole yhtä akuutti. D²TCP:n ensisijaisena käyttökohteena ovat OLDI-applikaatiot (‘Online Data Intensive Applications', esim. search, verkkokaupat, mainokset) joiden suoritukselle on asetettu tiukat aikarajat. D²TCP ei edellytä kytkimien laitteiston muokkaamista ja on vain päivitys Internetin TCP:sta mikä mahdollistaa sen käytön myös Internetissä. Kirjoittajien suorittaman D²TCP:n simulaation perusteella D²TCP suoriutuu aikarajojen täyttämisestä 75% DCTCP:tä paremmin ja 50% D³:sta paremmin (toinen aikarajat huomioiva ruuhkanhallinta-algoritmi).

D²TCP:n idea on säädellä ruuhkaikkunaa perustuen ruuhkan suuruuteen ja aikarajoihin. Jokaisesta datavirrasta on käytännössä mahdotonta pitää ajantasaista tietoa ja tämän vuoksi joudutaan tekemään päätöksiä perustuen rajalliseen informaatioon. Jokainen D²TCP-lähettäjä säätelee ruuhkaikkunaansa tietämättä siitä kuinka moni muista datavirroista on ruuhkautunut. Lopputuloksena on että ne datavirrat joilla on myöhempi aikaraja eivät valtaa kaistaa ja ne joilla on aikaisempi aikaraja saavat suuremman prioriteetin.

D²TCP on siinä mielessä edullinen että se edellyttää ainoastaan että kytkimet tukevat ECN:ää (yleisesti ottaen tämän päivän datakeskuksissa kytkimet tukevat ECN:ää). DCTCP:n tavoin D²TCP arvioi keskimääräistä ruuhkaa muuttujalla $a=(1-g)*a+g*f$ jossa f on se osa paketeista jotka oli merkitty edellisen ruuhkaikkunan lähetyksen aikana ja g on painoarvo joka on annettu uusille datanäytteille. Tämän lisäksi D²TCP ylläpitää muuttujaa d joka määrittää kuinka lähellä datavirran aikaraja on; mitä suurempi d, sitä lähempänä aikarajan erääntyminen. Ruuhkaikkunan kokoon vaikuttaa $p=a^d\,(a\leqslant 1, p\leqslant 1)$ (penalty function) joka on alun perin tarkoitettu grafiikkaohjelmointiin värin säätämiseen. Kun muuttujan p arvo on saatu, ruuhkaikkuna w määritellään seuraavasti:

$$w = w \times (1 - \frac{p}{2}),\ p > 0,$$

$$w = w + 1,\ p = 0 \quad .$$

Jos a=0 (toisin sanoen yhtään datapakettia ei ole merkitty ja näin ollen ruuhkaa ei ole) ja siten p=0, ruuhkaikkunaa kasvatetaan yhden datasegmentin verran kuten TCP:ssä. Jos kaikki datapaketit on merkitty, a=1 ja myös p=1 jolloin ruuhkaikkuna puolitetaan TCP:n tavoin. Jos a:n arvo on 0:n ja 1:n välissä, ruuhkaikkunan kokoa säätelee muuttuja p. Datavirroilla joilla aikaraja on lähempänä on pienempi p:n arvo ja datavirroilla joilla aikaraja on kauempana on korkeampi p:n arvo.

Jos p=a, D^{2}TCP käyttäytyy DCTCP:n tavoin. Periaatteessa DCTCP on D^{2}TCP:n erikoistapaus jossa kaikilla datavirroilla d=1 koko ajan joten kaikki nämä datavirrat reagoivat ruuhkaan samalla tavalla riippumatta aikarajoista.

Aikarajan läheisyyttä arvioiva muuttuja määritellään $d = \frac{T}{D}$ jossa T on tarvittava aika kyseiselle datavirralle lähettää kaikki data siinä tilanteessa jos aikarajoja ei huomioitaisi; D kertoo kuinka paljon on aikaa siihen kunnes kyseinen aikaraja menee umpeen. D^{2}TCP tekee tiettyjä oletuksia ruuhkan suhteen jonka perusteella T voidaan approksimoida antamalla sille arvo $T = B \times (\frac{3}{4} \times w)$ jossa w on datavirran sen hetkinen ruuhkaikkunan koko ja B kertoo kuinka paljon tavuja datavirralla on vielä lähettämättä. Edellistä tarkempi approksimaatio T:lle löytyy lähdeartikkelista [VHV12].

Muuttuja d:lle on asetettu yläraja (2.0) sitä varten ettei useampi datavirta joilla on tiukka aikaraja yrittäisi samaan aikaan väkisin lähettää dataa aggressiviisesti johtaen ruuhkautumiseen. Tällä tavalla tiukat aikarajat ikään kuin pyöristetään alaspäin ruuhkan välttämiseksi. Ruuhkan ollessa äärirajoillaan D^{2}TCP alkaa käyttäytymään kuin normaali TCP.

Tiivistettynä D^{2}TCP:n toiminta on seuraava: jos lähettäjä ei törmää merkittyihin paketteihin, ruuhkaikkunaa kasvatetaan yhden datasegmentin verran. Muussa tapauksessa lähettäjä laskee keskimääräisen ruuhkan, tarvittavan ajan lähettää loput datavirran datasta ja datavirran aikarajan läheisyyden. Tämän jälkeen ruuhkaikkunaa kasvatetaan perustuen funktioon p. Normaalit TCP:n ominaisuudet kuten hidas aloitus ja paketin hukkumisen aiheuttama aikakatkaisu ja uudelleenlähetys pysyvät samoina.

5.5 pFabric

Alizadeh et alin [AYS13] pFabric on minimalistinen ruuhkanhallinta-algoritmi joka perustuu näkemykseen että lähetysnopeuden hallinta on liian tehoton datavirtojen skedulointiin jonka vuoksi datavirtojen hallinta (flow control) ja lähetysnopeuden hallinta tulisi erottaa toisistaan ja suunnitella kukin yksitellen. Pfabricissa kytkimet voivat olla tavallista pienempiä ja kytkimet päättävät otetaanko saapuva paketti jonottamaan vai hylätäänkö paketti. Lähettäjä koodaa jokaisen paketin otsakkeeseen numeron joka kertoo tämän paketin / datavirran prioriteetista. Kun paketti saapuu kytkimelle puskurin ollessa täynnä, jos paketin prioriteetti on pienempi kuin kaikkien puskurissa jo jonottavien pakettien prioriteetti, paketti hylätään; jos taas puskurissa on paketti jolla on saapuvaa pakettia pienempi prioriteetti, kyseinen pienemmän prioriteetin paketti pudotetaan ja saapuva paketti päästetään puskuriin jonottamaan. Kun kytkimestä edelleenlähetetään paketti, valitaan paketti jolla on isoin prioriteetti ja joka on pisimpään jonottanut. Prioriteetti voidaan määritellä halutulla tavalla, se voi olla esim. aikaraja tai datavirran lähettämättä olevan datan määrä. Lähetysnopeuden hallinta on minimaalista ja sitä nopeutta kasvatetaan ainoastaan jos ilmenee että dataa on hävinnyt suuri määrä. Kadonneisiin paketteihin reagoidaan aikakatkaisun jälkeen kuten TCP:ssa.

5.6 TCP-CWR

Qin et al. [QSS13] esittävät matemaattisen mallin TCP Outcastille ja tämän perusteella ratkaisun TCP-CWR (TCP Congestion Windows Replacement) jossa lähettäjät muokkaavat ruuhkaikkunaa niin että se on kaikkien lähettäjien kesken sama. Lähetetettyään kokonaisen datalohkon lähettäjä lähettää ruuhkaikkunansa koon vastaanottajalle. Vastaanottaja laskee kaikkien lähettäjien keskimääräisen ruuhkaikkunan koon ja lähettää tämän jokaiselle lähettäjälle. Vastaanottaja ei pyydä seuraavaa datalohkoa palvelimilta ennen kuin on saanut näiltä tiedon kunkin ruuhkaikkunan koosta. Vastaanottaja laskee lähettäjille yhteisen ruuhkaikkunan jokaisen N lähettäjän erillisen ruuhkaikkunan $cwnd_i$ perusteella $cwnd = (\sum cwnd_i)/N$. Vastaanotettuaan tämän lähettäjät asettavat sen ruuhkaikkunansa kooksi ja jatkavat datapakettien lähetystä normaalisti.

5.7 TIMELY

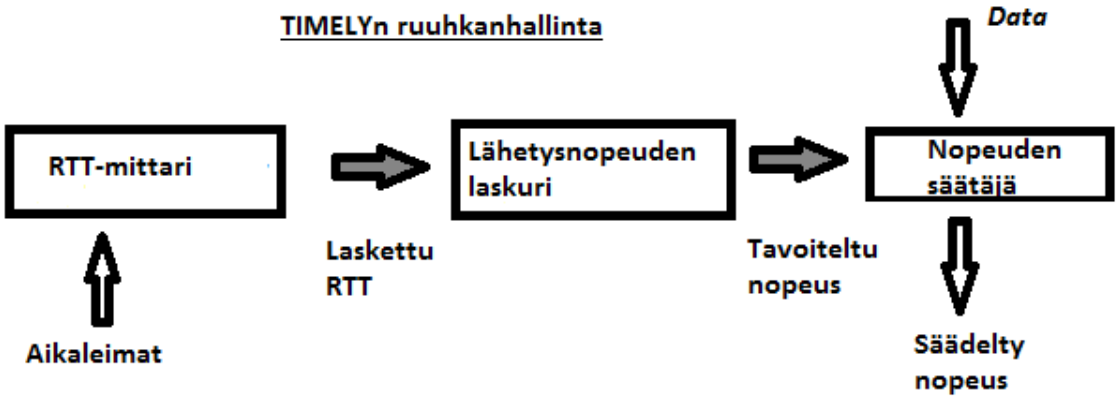
Mm. TCP-osiossa esitelty TCP Vegas on esimerkki viiveeseen perustuvasta algoritmista jossa ruuhkan merkki on latenssi eikä datapakettien häviäminen. DCTCP:ta taas ei suunniteltu tarkkailemaan latenssia koska tarkka datapakettien jonotusviiveen mittaaminen nähtiin liian työläänä tehtävänä. DCTCP:n tapaiset ECN:aan perustuvat algoritmit käyttävät kytkimen puskurin täyttymistä ruuhkan mittarina.

TIMELY (Transport Infomed by Measurement of Latency) [MLD15, ZGM16] perustaa logiikkansa Vegasin tavoin RTT-mittauksiin, motivaationa se että nykypäivän NIC:it (Network Interface Controller) mahdollistavat tarkan pakettien aikaleimaamisen datakeskuksissa ilman että erilaiset ympäristötekijät aiheuttavat häiriötä latenssin mittaamiselle.

TIMELY muokkaa lähetysnopeutta järjestelmällä joka koostuu kolmesta komponentista; 1) ruuhkaa tarkkailevasta RTT-mittarista (RTT measurement engine); 2) lähetysnopeuden laskurista (rate computation engine) ja 3) nopeuden säätäjästä (pacing engine) joka määrittelee lähetysnopeuden lisäämällä viiveitä lähetettyjen datasegmenttien väliin saavuttaakseen tavoitellun nopeuden. RTT-mittari laskee RTT:n datasegmenttiä kohden (TCP:ssa RTT mitataan 1-2 pakettia kohden) ja RTT mitataan yksinomaan etenemisviiveen ja (kytkimissä tapahtuvan) jonotusviiveen tuotoksena. Jos t_{send} on aika siitä kun ensimmäinen paketti on lähetetty ja $t_{completion}$ on aika jolloin koko segmentistä on lähetetty ACK, TIMELY laskee RTT:n seuraavasti: $RTT = t_{completion} - t_{send} - \frac{seg}{NLR}$ jossa seg on segmentin koko ja NLR NICin maksiminopeus.

Aina kun on saatu ACK kokonaisesta datasegmentistä, RTT-mittari laskee RTT:n mikrosekunnin tarkkuudella (TIMELY olettaa että NICit pystyvät tähän), luovuttaa lasketun RTT:n lähetysnopeuden laskurille ja tästä lasketaan datavirralle tavoiteltu nopeus joka taas syötetään nopeuden säätäjälle. Lähetysnopeuden laskuri päivittää tavoitellun nopeuden aina kun kokonaisesta segmentistä on saatu ACK. Kuvassa 5.7.1 on esitelty TIMELY-frameworkin rakenne.

Timely laskee viivegradientin (delay gradient) joka ilmaisee kuinka nopeasti kytkimellä pakettijono kasvaa tai pienenee. Gradienttia käytetään virhemittarina jonka perusteella



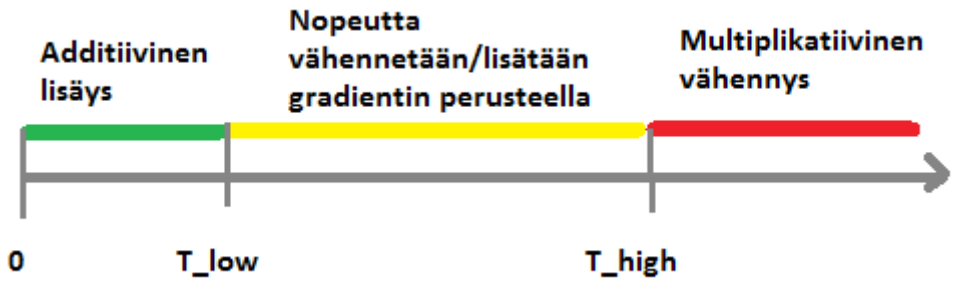
Kuva 5.7.1) TIMELY-ruuhkanhallinnan rakenne [MLD15]

yksittäisen yhteyden lähetysnopeutta säädetään.

Jos pullonkaulalinkille lähettää samanaikaisesti N lähettäjää siten että kokonaislähetysnopeus on $y(t)$, gradientti lasketaan kaavalla $\frac{y(t)-C}{C}$ jossa C on linkin/pakettijonon suurin mahdollinen nopeus jolla jono tyhjenee (drain rate). Gradientti on positiviinen jos $y(t)>C$ jolloin gradientti ilmaisee kuinka nopeasti jono kasvaa; negatiivinen gradientti (jos $y(t)<C$) ilmaisee kuinka nopeasti jono tyhjenee. Jos jono pysyy muuttumattomana tai jono on tyhjä, gradientti on 0. TIMELY pyrkii pitämään *y(t)*:n ja *C*:n samana. Jos $g\leqslant 0$, linkillä on tilaa suuremmalle nopeudelle jolloin yksittäisen yhteyden nopeutta R kasvatetaan $R=R+i$ missä i on ennalta määritelty kaistan lisäysvakio. Jos $g>0$, $y(t)$ on suurempi kuin linkin kapasiteetti ja yksittäistä nopeutta vähennetään gradientin skaalaamalla vakiolla *b*;

$R=R(1-bg)=R(1-b(\frac{y(t)-C}{C}))$. Yksittäisen yhteyden uusi lähetysnopeus lasketaan uudestaan aina kun kokonaisesta datasegmentista on saatu ACK.

TIMELY käyttää gradienttia nopeuden säätämisen motiivina vain silloin kun yhteyden RTT on kahden ennalta määritellyn kynnysarvon välissä (kuva 5.7.2). Tällä ratkaisulla pyritään hallitsemaan erikoistapaukset joissa joko pullonkaulalinkkiä hyödynnetään huomattavan tehottomasti tai latenssi on erityisen suurta. Jos RTT on alle alemman kynnysarvon T_{low} , lähetysnopeutta kasvatetaan $R=R+i$; jos RTT on T_{low} :n ja ylemmän kynnysarvon T_{high} välissä, lähetysnopeus määrittyy gradientin mukaisesti (mikä kuvattiin ylempänä); ja jos RTT on suurempi kuin T_{high} , lähetysnopeutta



Kuva 5.7.2: Kun TIMELYssa RTT on kahden kynnysarvon välissä, lähettäjän nopeus lasketaan viivegradientin perusteella [MLD15].

vähennetään multiplikatiivisesti $R = R\left(1 - b\left(1 - \frac{T_{high}}{RTT}\right)\right)$. TIMELYn datavirrat aloittavat nopeudella joka on suhteessa siihen miten paljon vastaanottajalle on menossa datavirtoja: jos linkillä on samanaikaisesti N datavirtaa, uusi datavirta aloittaa nopeudella $R = Capacity / (N + 1)$ (*Capacity*=linkin kapasiteetti).

5.8 MPTCP

Useimmat tässä tutkielmassa käsitellyistä ruuhkanhallinta-algoritmeista on suunniteltu hyödyntämään yhtä polkua verkon solmujen välillä mutta tilanteesta riippuen datakeskuksen topologia sallii useampia polkuja kahden solmun välillä (esimerkiksi Bcube ja FatTree). MPTCP (Multipath TCP) on TCP:n muunnos jonka ideana on hyödyntää vaihtoehtoisia polkuja solmujen välillä [BBR10]. Tätä kirjoittaessa IETF (Internet engineering task force) on julkaissut MPTCP-spesifikaation kokeellisena standardina (standardi julkaistiin 2013 tammikuussa).

MPTCP-yhteys muodostetaan kuten TCP:ssä eli 3-vaiheisessa kättelyssä jossa samalla määritellään tukeeko molempi yhteyden osapuoli MPTCP:tä. Lähettäjä lähettää SYN-segmentin jossa on määritelty MPC-optio (Multi Path Capable) joka kertoo että lähettäjä haluaa käyttää MPTCP:tä datan siirrossa. Myönteisessä tapauksessa vastaanottaja vastaa SYN/ACK-segmentillä joka myös sisältää MPC-option. Molemmissa viesteissä lähettäjä (vastaanottaja) lähettää myös oman uniikin avaimensa joista saadaan hash-funktiolla uniikki poletti (token) lähettäjälle (tai vastaanottajalle). Lopulta lähettäjä tai vastaanottaja saa tietää kohteena olevan palvelimen vaihtoehtoisista IP-osoitteista jonka jälkeen voidaan muodostaa lisää alidatavirtoja (subflow) lähettäjän ja vastaanottajan välillä (jotka ovat osa yhtä suurempaa datavirtaa). Jatkossa kun tässä

tekstissä puhutaan yhden datavirtan hajauttamisesta lähettäjän ja vastaanottajien välissä olevien vaihtoehtoisten polkujen kesken, käytetään nimitystä alidatavirta.

Aloitettaessa uusi alidatavirta lähettäjä tai vastaanottaja lähettää SYN/JOIN-viestin joka kertoo toiselle osapuolelle että uusi alidatavirta on osa jo olemassa olevaa MPTCP-yhteyttä. Samassa viestissä lähetetään myös kohteelle kuuluva uniikki poletti jonka perusteella jokainen alidatavirta voidaan sitoa sitä vastaavaan (useista alidatavirroista koostuvaan) MPTCP-yhteyteen. Useampi alidatavirta joka kulkee saman lähettäjän ja vastaanottajan välillä voi käyttää samoja IP-osoitteita mutta siten että data kulkee eri porttien kautta; vaihtoehtoisesti voidaan käyttää eri IP-osoitteita lähettäjän ja vastaanottajan välillä. Jos dataa häviää matkalla, se voidaan uudelleenlähettää toista polkua pitkin.

Topologioissa joissa solmujen välillä kulkee useita polkuja saatetaan tarvita datan lähettäjästä erillinen keskitetty reitityskoneisto sillä lähettäjän ei voida olettaa tietävän vaihtoehtoisten polkujen kuormituksesta mitään. Tällaisen keskitetyn datavirtojen järjestelijän käytössä on ongelmana että sillä on samaan aikaan kiire muiden tehtävien kanssa, esimerkiksi tilastojen keräämisessä. Yksinkertaisin ratkaisu reititykseen on RLB (Random Load Balancing) jossa jokaiselle alidatavirralle arvotaan polku jota pitkin alidatavirta lähetetään. RLB:n käytännön toteutus voidaan järjestää usealla tavalla tyypillisissä tämän päivän kytkimissä, esimerkiksi syöttämällä datapaketin otsakkeen kenttiä hash-funktiolle jonka tulos määrittää polun jota pitkin alidatavirta lähetetään. Usein kuitenkin tällainen ratkaisu ei käytännössä onnistu hyödyntämään tehokkaasti solmujen välistä kaistaa; jotkut alidatavirrat satunnaisesti valitsevat saman polun kun taas jotkut linkit jäävät satunnaisesti kokonaan käyttämättä. RLB:n kanssa onkin usein käytössä keskitetty datavirtojen järjestelijä. RLB-tyyppinen ratkaisu alidatavirtojen järjestelyssä toimii hyvin jos datakeskuksen tietoliikenne koostuu suuresta joukosta lyhyitä TCP-yhteyksiä. Pitkät ja korkeaa läpisyöttöä vaativat yhteydet kuitenkin ovat myös yleisiä datakeskuksessa [PB14].

TCP:ssä pakettien järjestämiseen järjestysnumeroilla riittää yksi numeroavaruus. Näin voitaisiin menetellä myös MPTCP:ssä mutta tämän seurauksena paketit sekoittuisivat päästessään perille ja ne jouduttaisiin järjestämään uudelleen. Koska pakettien uudelleenjärjestäminen on tavallisesti merkki siitä että paketteja on hävinnyt, tarvitaan erityisiä algoritmeja erottamaan milloin on kyse MPTCP:n pakettien uudelleen

järjestämisestä ja milloin kyse on pakettien häviämisestä. Jos pakettien järjestämisessä käytetään vain yhtä numeroavaruutta, on vaikeaa päätellä mitä polkuja pitkin data on saapunut [BBR10, PB14]. Jotta eri alidatavirrat eivät sekoittuisi toisessa päässä kun ne kootaan yhteen kokonaiseksi datasegmentiksi, jokainen alidatavirta ylläpitää kahta eri numeroavaruutta järjestysnumeroille; toinen numeroavaruus kertoo kyseisen alidatavirran datapakettien keskinäisen järjestyksen ja toinen numeroavaruus kertoo järjestyksen suhteessa muihin alidatavirtoihin kun ne kootaan yhteen vastaanottajan päässä. Jokainen MPTCP:n alidatavirta käyttää TCP:n tyylistä AIMD-protokollaa (additive increase, multiplicative decrease) reagoidessaan ACKeihin ja pakettien häviämiseen mutta MPTCP kuitenkin muokkaa additiivista lisäystä:

- jos r on jokin MPTCP-yhteyden alidatavirta, ja w_r kyseisen alidatavirran ruuhkaikkuna, jokaisen ACKin vastaanottamisen yhteydessä w_r:ää kasvatetaan arvolla $min(\frac{a}{w}, \frac{1}{w_r})$ jossa w summaa yhteen kaikkien alidatavirtojen w_r ruuhkaikkunoiden arvot ja muuttuja a on IETF-spesifikaation mukaan laskettava muuttuja.
- Datapaketin häviämisen yhteydessä ruuhkaikkuna w_r puolitetaan (kuten TCP:ssä)

Eräs haaste MPTCP-protokollaa toteutettaessa on päättää miten reagoida tilanteisiin joissa useampi alidatavirta osuu samalle pullonkaulalinkille. Jotkut ratkaisut turvautuvat ulkoiseen mekanismiin välttääkseen tämän ja jotkut yksinkertaisesti jättävät asian huomiotta.

Sovellusten ei tarvitse olla tietoisia siitä että tietoliikenne käyttää nimenomaan MPTCP:tä eikä tavallista TCP:tä, toisin sanoen sovelluksia ei tarvitse modifioida MPTCP:n käyttöä varten. Kehittyneemmät sovellukset voivat kuitenkin hyödyntää erityistä socket API:a määrätäkseen miten MPTCP-yhteys jakautuu eri alidatavirtojen kesken.

MPTCP:n hyödyllisyys datakeskusten käytössä on selkeä: kun alidatavirtojen määrä kasvaa, verkon läpisyöttö kasvaa myös. Datakeskuksen topologia vaikuttaa siihen kuinka paljon alidatavirtoja tarvitaan per MPTCP-yhteys. Raiciu et al. [RPB10] ovat verranneet MPTCP:n toteutusta kolmessa topologiassa. Samassa artikkelissa kirjoittajat

vertaavat RLB:n toimivuutta sekä keskitetyn datavirtojen järjestelijän käyttöä omaan linked increase-algoritmiinsa [RWH]. Tuloksista päätellen järjestelijän toimivuus riippuu siitä kuinka usein se ajetaan.

Olteanu ja Raiciu [OR16] sanovat että datakeskusten käytössä MPTCP-liikenne tarvitsee erityisen load balancerin ja esittävät oman ratkaisunsa tähän. Datakeskuksessa ajettavalla palvelulla on yksi julkinen osoite ja tälle ylläpitäjä on määritellyt joukon yksityisiä osoitteita ja load balancerin tehtävä on jakaa julkiseen osoitteeseen matkalla oleva datalähetys eri yksityisten osoitteiden kesken. Yksi MPTCP-yhteys pitää sisällään mielivaltaisen määrän alidatavirtoja joista jokainen näyttää erikseen tarkasteltuna itsenäiseltä yksittäiseltä TCP-yhteydeltä ja vain yhteyden SYN-paketeissa on tietoa jolla voi erottaa eri yhteydet toisistaan. Internetin käytössä load balancer toimii niin että se syöttää datapaketin kenttiä hash-funktiolle ja funktion tuloksen perusteella päätetään mille palvelimelle paketti lähetetään. Jos tätä sovelletaan MPTCP-yhteyden eri alidatavirroille, nämä ohjautuvat väärille palvelimille sillä yhden MPTCP-yhteyden jokainen alidatavirta kuuluisi ohjata samaan yksityiseen osoitteeseen mutta tyypillinen load balancer lähettää helposti kaikki paitsi ensimmäisen alidatavirran eri yksityisiin IP-osoitteisiin. Olteanun ja Raiciun ratkaisussa tokenit (jotka määrittävät mihin MPTCP-yhteyteen kukin alidatavirta kuuluu) luodaan siten että multiplexerin annetaan päättää uusien MPTCP-yhteyksien avaimet (joista yhteyksien uniikit poletit johdetaan) ja sulauttamalla tokenit joka paketin aikaleimaan.

5.9 XMP

Cao et al. [CFX13] esittävät vaihtoehtoisia polkuja hyödyntävän algoritmin jossa suuret datavirrat käyttävät MPTCP:tä ja pienet datavirrat käyttävät TCP:tä. XMP:ssä (explicit multipath control) linkkien kuormitusta hallitaan ECN-mekanismia muokkaavalla BOS (Buffer Occupancy Suppression)-algoritmilla ja TraSh (Traffic Shifting) algoritmi siirtää datavirtoja ruuhkautuneilta poluilta vähemmän ruuhkautuneisiin ja näin tehostaa suurien datavirtojen läpisyöttöä.

BOS-algoritmin rakenne on pääpiirteissään seuraava. Kun (ECN:ää tukevalle) kytkimelle osuu paketti, se merkitään jos kytkimen puskurin jonon (varsinainen, ei arvioitu) koko ylittää kynnysarvon *K*. Merkityn paketin saapuessa lopulta vastaanottajalle vastaanottaja sisällyttää ECE (ECN Echo)- ja CWR (congestion

window reduced)-signaalin lähettäjälle lähetetyssä ACKissa. Algoritmissa on TCP:lle tuttuun tapaan kaksi vaihetta, hidas aloitus ja ruuhkanvälttely. Hitaan aloituksen aikana jos lähettäjä saa ACKin jossa sekä ECE että CWR ovat molemmat 0, lähettäjä kasvattaa ruuhkaikkunaa yhdellä ja muussa tapauksessa siirtyy ruuhkanvälttelyyn. Ruuhkanvälttelyn aikana ruuhkaikkunaa vähennetään $1/b$ (b vakio) verran jos lähettäjä vastaanottaa ACKin jossa ECE ja CWR eivät molemmat ole 0; muussa tapauksessa ikkunaa kasvatetaan muuttujan d verran. Ikkunaa kasvatetaan tai vähennetään vain kerran yhden kierroksen aikana; kierros on tässä aika joka kuluu tietyn paketin lähettämisestä siihen kun sen ACK on vastaanotettu (yksityiskohdat löytyvät lähdeartikkelista).

Jos (s,r) on MPTCP-yhteyteen s kuuluva alidatavirta joka kulkee polun r kautta, kokonaisuudessaan s:n lähetysnopeus on $y_s = \sum x_{s,r}$ jossa $x_{s,r}$ on nopeus jolla s lähettää polkua r pitkin eli (s,r):n lähetysnopeus. Muuttuja $d_{s,r}$ säätelee alidatavirran (s,r) ruuhkaikkunaa BOS-algoritmin mukaisesti. Kirjoittajat laskevat s:n kokonaiselle lähetysnopeudelle y_s utility-funktion jonka perusteella he johtavat $d_{s,r}$:lle tarkan arvon $d_{s,r} = \frac{T_{s,r} \times x_{s,r}}{T_s \times y_s}$ jossa $T_s = min(T_{s,r}, r \in R_s)$, $T_{s,r}$ on SRTT (Smoothed RTT) polulla r ja R_s koostuu kaikista poluista joita pitkin s lähettää dataa.

TraSh-algoritmin toiminta on seuraava. Jos t on ajanjakso algoritmin silmukassa ja $d_{s,r}(t)$ on $d_{s,r}$:n arvo ajanhetkellä t, hetkellä $t=0$ datavirta s asettaa $d_{s,r} = 1 \quad \forall r \in R_s$ jonka jälkeen s siirtyy vaiheeseen jossa alidatavirran nopeus suppenee kohti tiettyä arvoa (Rate Convergence). Kyseisessä vaiheessa ajanhetkellä t jokainen (s,r) ajaa BOS-algoritmia kunnes jokaisen (s,r):n nopeus $x_{s,r}$ suppenee arvoon $x_{s,r}(t) = \frac{b \times d_{s,r}(t) \times (1 - p_{s,r}(t))}{T_{s,r} \times p_{s,r}(t)}$ missä $p_{s,r}$ on todennäköisyys sille että edes yksi alidatavirran (s,r) paketti on ECN-merkitty yhden kierroksen aikana. Nopeuden suppenemisen jälkeen s laskee kokonaisnopeuden $y_s(t) = \sum x_{s,r}(t), r \in R_s$ ja $T_s = min(T_{s,r}, r \in R_s)$ ja säätää seuraavalle ajanjaksolle $(t+1)$ muuttujan $d_{s,r}$ jokaisella (s,r):

$d_{s,r}(t+1)=\frac{T_{s,r}\times x_{s,r}(t)}{T_s\times y_s(t)}$. Seuraavalla ajanjaksolla t TraSh-algoritmi aloittaa uuden iteraation alkaen vaiheesta jossa alidatavirtojen nopeudet suppenevat tiettyä arvoa kohti.

Kokonaisuudessaan XMP toimii näin: jokainen alidatavirta (s,r) muokkaa lähetysikkunaansa BOS-algoritmilla ja muuttujan d määrittää TraSh-algoritmi. Parametrit joilla voidaan vaikuttaa algoritmin toimintaan ovat ruuhkaikkunaa säätelevä muuttuja b sekä kytkimen puskurin kynnysarvo K.

5.10 DCMC

Suurin osa tässä tukielmassa käsitellyistä ruuhkanhallinta-algoritmeista on suunniteltu unicast-liikenteeseen jossa datan lähettäjiä on monta mutta vastaanottajia vain yksi. Monilähetyksestä (multicast) syntyvää liikennettä (yksi lähettää monelle tai moni lähettää monelle) kuitenkin on myös runsaasti datakeskuksissa mutta monilähetyksen ruuhkanhallinnan tutkimus on ollut vähäisempää kuin unicastin. Unicast-liikenteessä voi olla yksi pullonkaulalinkki mihin ruuhka keskittyy mutta monilähetyksessä pullonkaulalinkkejä voi olla useampia.

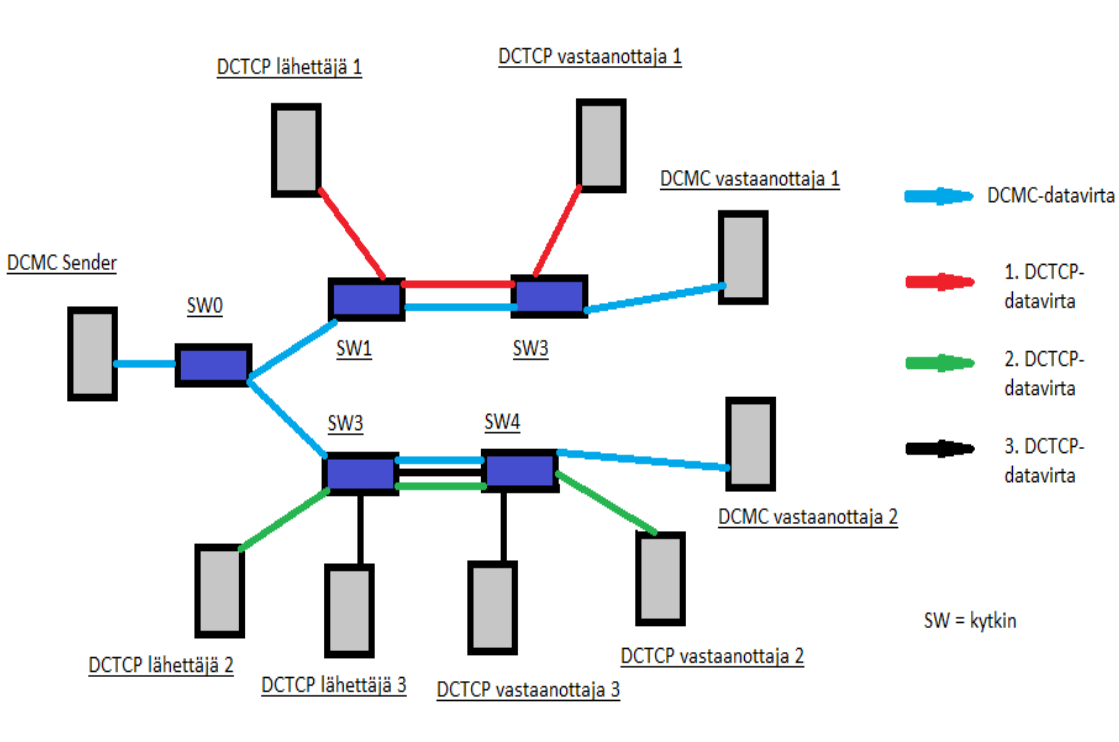
DCMC (Data Center Multicast Congestion Control) [AMY16] yrittää torjua multicastista syntyvää ruuhkaa ja on suunniteltu niin että se pystyy toimimaan samaan aikaan DCTCP-datavirtojen kanssa (DCTCP on unicast-protokolla). Monilähetys käyttää yleisesti UDP:ta kuljetuskerroksen protokollana ja UDP ei yleensä suorita minkäänlaista ruuhkanhallintaa ja näin ollen kun UDP ja TCP kulkevat samaa reittiä pitkin, UDP oletusarvoisesti käyttää ahneesti kaistaa ottaen sitä samalla pois TCP:ltä joka reiluna protokollana pienentää lähetysnopeutta ruuhkatilanteessa.

DCMC perustuu WAN (Wide Area Internet)-ympäristössä käytettyyn monilähetyksen ruuhkanhallinta-algoritmiin TFMCC:hen (TCP-friendly Multicast Control) joka säätelee lähetysnopeuttaan perustuen arvioituun TCP:n läpisyöttöön joka lasketaan datan vastaanottajilta saadusta palautteesta. TFMCC:ssa vastaanottajat mittaavat RTT:ta ja miten paljon datapaketteja on hävinnyt arvioidakseen vastaanottajalle saapuvan

nopeuden kaavalla $T_{TCP} = \frac{s}{t_{RTT} * (\sqrt{\frac{2p}{3}} + (12 * \sqrt{\frac{3p}{8}}) * p * (1 + 32 * p^2))}$, missä t_{RTT} on RTT, p on pakettien häviämisnopeus ja s on paketin koko. Vastaanottaja lähettää arvioidun nopeuden lähettäjälle joka vertaa tätä omaan lähetysnopeuteensa. Jos arvioitu nopeus on suurempi kuin lähettäjän varsinainen nopeus, lähettäjä kasvattaa varsinaista nopeuttaan ja muussa tapauksessa muuttaa varsinaisen nopeuden samaksi kuin arvioitu nopeus. Lähettäjä valitsee vastaanottajista sen johon arvioitu lähetysnopeus on huonoin kaikista vastaanottajista ja tälle vastaanottajalle annetaan nimi CLR (current limiting receiver). CLR lähettää palautetta lähettäjälle joka RTT:n välein. Lähettäjä lähettää CLR:n arvioidun lähetysnopeuden kaikille (monilähetyksen) vastaanottajille joista jokainen (poislukien CLR) lähettää palautetta lähettäjälle vain silloin kun arvioitu nopeus tämän vastaanottajan ja lähettäjän välillä on alle vastaavan yhteyden varsinaisen nopeuden tai edellisestä palautteen lähetyksestä on kulunut tietty aika; tällä tavoin vältetään turhia palautteita.

DCMC eroaa TFMCC:sta siten että DCMC ei pyri kanssaelämään TCP-datavirtojen kanssa vaan DCTCP-virtojen kanssa (kuva 5.10); DCMC-vastaanottaja laskee sille saapuvan DCTCP-nopeuden. Kuten DCTCP:ssä, DCMC päättelee ruuhkautumisen siitä kun kytkimellä jonottavien pakettien määrä ylittää ennalta määritellyn kynnysarvon (TFMCC päättelee saman siitä kun datapaketti häviää). Kuten TFMCC:ssä, DCMC:ssä lähettäjä valitsee vastaanottajan johon arvioitu lähetysnopeus on pienin kaikista näistä vastaanottajista jonka jälkeen lähettäjä asettaa lähetysnopeudekseen tämän saman vastaanottajan arvioidun DCTCP-nopeuden. Kuten DCTCP, DCMC laskee merkittyjen pakettien keskimääräisen osuuden kaikkien pakettien yhteenlasketusta määrästä funktiolla $a = (1-g) * a + g * F$ jossa F on merkittyjen pakettien osuus edellisessä algoritmin iteraatiossa ja g on painotettu muuttuja (=1/16). Muuttuja a päivitetään joka RTT:n välein; a lähellä nollaa viittaa vähäiseen ruuhkaan ja a lähellä ykköstä viittaa suureen ruuhkaan. Pakettien häviämisnopeus p ja RTT mitataan kuten TFMCC:ssä.

DCMC määrittelee CE-ajanjakson joka on kahden peräkkäisen ECN-ruuhkasignaalin (eli merkityn paketin vastaanoton) välinen aika. Jos Y_i on tällä ajanjaksolla lähetettyjen pakettien määrä ja A_i on tämän ajanjakson pituus, arvio



Kuva 5.10: Yksi DCMC-lähettäjä monilähettää kahdelle vastaanottajalle; matkalla DCMC:n datavirrat jakavat pullonkaulalinkin useaan kertaan eri DCTCP-datavirtojen kanssa [AMY16].

lähetysnopeudelle johdetaan *Y*:n ja *A*:n odotusarvojen *E[Y]*:n ja *E[A]*:n osamäärästä. Tarkka funktio arvioidun nopeuden laskemiseksi löytyy lähdeartikkelista.

5.11 M21TCP-A

M21TCP-A:ssa [AM15] datapaketin saapuessa kytkimelle saapuvan paketin TCP- tai IP-otsakkeeseen sulautetaan kenttä joka kertoo kuinka monta konetta lähettää samanaikaisesti dataa saman kytkimen kautta. Lähettäjä laskee tämän perusteella ruuhkaikkunansa maksimin siten että jos kaikki lähettäjät lähettävät samanaikaisesti, kytkimellä riittää tilaa kaikkien lähettäjien paketeille.

Kun kytkimelle saapuu paketti, reititin tarkastaa paketin parametrit jotka yksilöivät tämän datavirran; jos kytkimen muistissa ei ole valmiiksi datavirtaa jolla on saapuneen paketin parametrit, reititin tallentaa uuden datavirran muistiin. Parametreihin kuuluvat lähtö- ja kohdeosoitteet, lähtö- ja kohdeportit sekä käytettävän protokollan nimi. M21TCP-A olettaa että lähettäjä pystyy sisällyttämään lähettämiinsä datapaketteihin ylimääräisen 32 bittisen TCP- tai IP-kentän. Saadessaan paketin kytkimeltä

vastaanottaja lähettää kytkimen laskeman lähettäjien / datavirtojen määrän kaikille lähettäjille (jotka lähettävät saman kytkimen kautta) ACK-paketin välityksellä. Kun lähettäjän ja vastaanottajan välillä on useampi kytkin, kytkin voi koodata vastaanotettuun pakettiin lähettäjien määrän ainoastaan jos kytkimessä jo tallennettu lähettäjien määrä on suurempi kuin saapuneessa paketissa ilmoitettu. Lähettäjien määrä lasketaan joka RTT:n kulumisen jälkeen. Lähettäjä asettaa ruuhkaikkunansa koon seuraavasti: $MaxWind = \frac{B-(MHS \times N)}{N}$ jossa B on kytkimen pakettipuskurin koko, MHS on datapaketin otsakkeen koon minimi ja N kytkimen laskema lähettäjien määrä.

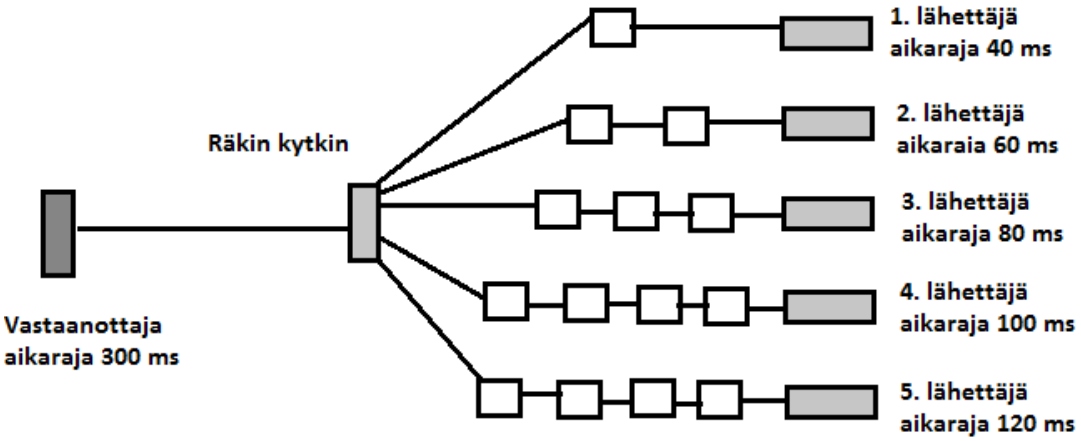
M21TCP-A ei periaatteessa ole ruuhkanhallinta-algoritmi itsessään vaan se on tarkoitettu käytettäväksi muiden ruuhkanhallinta-algoritmien kanssa; M21TCP-A ainoastaan määrittää ruuhkaikkunalle maksimin eikä muuten muuta alla olevaa ruuhkanhallinta-algoritmia. Jos C on funktio jolla ruuhkanhallinta-algoritmi laskee ruuhkaikkunalle uuden koon, M21TCP-A:ssa lähettäjän ruuhkaikkuna lasketaan seuraavasti: $cwnd = min(C, MaxWind)$.

M21TCP-A on tarkoitettu ainoastaan yhden polun TCP:n käyttöön, toisin sanoen se on tarkoitettu käytettäväksi topologioissa joissa lähettäjien ja vastaanottajan välillä on vain yksi polku. Jos M21TCP-A:ta käytetään topologiassa jossa on vaihtoehtoisia polkuja isäntäkoneiden välillä, tarvitaan avustavia mekanismeja.

5.12 UCCS

UCCS (Urgency and Congestion Control Scheme) pyrkii torjumaan incastia ottaen D^{2}TCP:n tavoin huomioon datavirtojen aikarajat [TYP16]. UCCS ylläpitää listaa joka sisältää jokaista TCP-yhteyttä vastaavan olion. Jokaisesta oliosta tallennetaan vastaavan datavirran koko, aikaraja ja datavirran kulkeman polun pituus (eli kuinka monen kytkimen yli data kulkee). Algoritmi jakautuu kahteen osaan; toinen määrittelee datavirtojen järjestyksen eli prioriteetin ja toinen säätää datavirtojen lähetysnopeutta perustuen aikarajaan ja datavirran kokoon.

UCCS laskee datavirran toimitusajan kaavalla $T_{flow} = T_{pro} + T_{trans} + T_{queue} + T_{proc}$ eli toimitusaika on etenemisajan (kuinka kauan digitaalinen signaali matkaa lähettäjältä vastaanottajalle), siirtoajan (kuinka kauan paketti kulkee linkin yli joka on suhteessa



Kuva 5.12: UCCS: Jokaisen lähettäjän lähettämällä datalohkolla on eri aikaraja ja lähettäjät sijaitsevat eri etäisyyksien päässä vastaanottajasta (valkoiset neliöt kuvassa ovat kytkimiä) jonka vuoksi datan lähetys eri lähettäjiltä kestää eri ajan [TYP16].

paketin kokoon), jonotusajan (kuinka kauan paketti jonottaa kytkimen puskurissa) ja prosessointi-ajan (kuinka kauan reititin käsittelee paketin otsakkeen) summa. Toimitusaika riippuu siitä kuinka monen kytkimen kautta data kulkee lähettäjältä vastaanottajalle. Datavirran kiireellisyys ja jäljellä oleva aika aikarajan erääntymiseen lasketaan kaavoilla

$$Urgency = \frac{(V_{flow_r} - V_{flow_p})}{Vflow_p}$$ ja $$Deadline_{flow_r} = Deadline_{flow} - \sum T_{flow_i}$$. V_{flow_p} on datan lähetysnopeuden odotusarvo ja V_{flow_r} on tarvittava lähetysnopeus jolla datavirta saadaan lähetettyä loppuun aikarajan erääntymiseen mennessä. Jos useamman datavirran kiireellisyys on sama, lasketaan muuttuja joka määrittää datavirran prioriteetin joka on suhteessa datavirran kulkeman matkan polun pituuteen

$$R_{flow} = \frac{Hop_{Num}}{Hop_{MaxNum}}$$ jossa Hop_{MaxNum} on pisin matka kahden koneen välillä koko datakeskuksessa ja Hop_{Num} datavirran kulkemalla polulla olevien linkkien määrä.

Toinen osa algoritmista pyrkii varmistamaan että kaikkien lähettäjien lähetysnopeuksien summa on pienempi tai yhtä suuri kuin pullonkaulalinkin kapasiteetti, toisin sanoen jos

$Rate_{flow-i}$ on datavirran numero i lähetysnopeus ja C on linkin kapasiteetti niin $\sum Rate_{flow-i} \leqslant C$. Jokainen datavirta on tallennettu listaan ja jokaisen prioriteetti sekä kaikkien prioriteettien summa lasketaan ja jokaiselle datavirralle lasketaan painoarvo. Datavirta *i*:n prioriteetti lasketaan kaavalla $Priority_i = Urgency_i + R_i$. Yksittäisen datavirran nopeus lasketaan kaavalla $Rate_{flow-i} = aBW$ missä BW on kaikkien lähettäjien ja vastaanottajaan johtavan räkin kytkimen välissä olevien linkkien pienin kapasiteetti; painoarvo lasketaan seuraavasti: $a = \frac{Priority_i}{\sum Priority_i}$.

5.13 OTCP

SDN (Software defined networking)-arkkitehtuuri mahdollistaa verkon liikenteen kontrolloimisen keskitetyn kontrollerin tai ohjelmiston kautta. Yleisin tällaisista protokollista on OpenFlow jonka avulla voidaan kerätä tietoa verkon ominaisuuksista kuten topologiasta, latensseista, läpisyötöstä ja kytkinten puskureiden koosta. OTCP (Omniscient TCP) [JPP16] käyttää OpenFlowta hyväksi laskien edellämainittujen

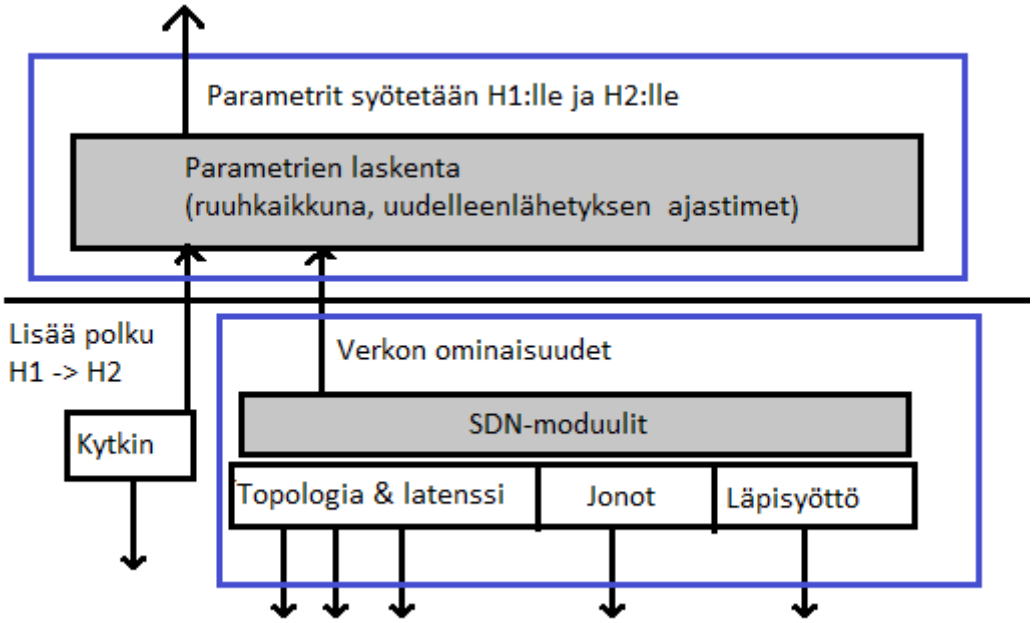
ominaisuuksien arvot ja tämän perusteella säätää uudelleenlähetyksen ajastimia ja ruuhkaikkunan koon siten että ruuhkaikkunan koko vastaa isäntäkoneiden välistä BDP:ta.

OpenFlow-verkon OFDP-protokollalla (OpenFlow Discovery Protocol) pystytään mittaamaan kahden isäntäkoneen välinen viive joka asetetaan uudelleenlähetyksen ajastuksen (RTO = Retransmission timeout) minimin arvoksi. Uudelleenlähetyksen ajastimen maksimi kahden koneen H_1 ja H_2 välisen polun välillä määritellään ajastimen minimin ja polulla olevien kytkimien jonotusviiveiden avulla :

$RTO_{max}(H_1 \rightarrow H_2) = RTO_{min}(polku\ R) + \sum(\frac{Q_s}{T_s})$ missä Q_s on kytkimen $s \in R$

jonon koko ja T_s on *s*:sta ulos menevä lähetysnopeus (kuinka nopeasti paketit lähetetään kytkimestä eteenpäin); $H_1 \rightarrow H_2$ on polku koneiden H_1 ja H_2 välillä ja R on polulla olevien kytkinten joukko. OpenFlow-protokolla pystyy keräämään nämä tiedot pakettijonojen ominaisuuksista ja lähetysnopeuksista.

Koneesta H_1 koneeseen H_2 menevän liikenteen ruuhkaikkunan koon maksimiksi



Kuva 5.13: OTCP-arkkitehtuuri [JPP16].

asetetaan koneiden välisen viiveen ja koneiden H_1 :n ja H_2 :n välillä olevien linkkien suurimman nopeuden tulo:

$CWND_{max}(polku\ H_1 \rightarrow H_2) = BDP_{polku\ H_1 \rightarrow H_2} = RTT_{polku\ H_1 \rightarrow H_2} \times T_{polku\ H_1 \rightarrow H_2}$ missä RTT on polun viive ja T on polun linkkien suurin nopeus. Alustava ruuhkaikkunan koko on murto-osa ruuhkaikkunan maksimikoosta koska $CWND_{max}$ on yhtä kuin suurin mahdollinen ruuhkaikkuna tilanteessa jossa polulla liikkuisi yksi datavirta (t.s. tilanteessa jossa polulla ei ole ruuhkaa); jos polulla on useampi datavirta, jokaisen datavirran alustavan ruuhkaikkunan koon tulisi olla yhtä kuin :

$CWND_{init} = min(1, \frac{CWND_{max}}{a})$ missä a on polulla olevien datavirtojen määrä. OTCP:ta voidaan käyttää DCTCP:n kanssa siten että OTCP määrittää sekä alustavan että suurimman ruuhkaikkunan koon ja lisäksi uudelleenlähetyksen ajastimet samalla kun DCTCP vahtii sitä etteivät kytkinten puskurit ylivuoda.

5.14 QCN

QCN (Quantized Congestion Notification) [THY12] on Ethernetin ruuhkanhallinta-mekanismi jossa kytkin ruuhkan tapahtuessa antaa palautetta lähettäjälle. Kytkin

tarkkailee pakettijonon kokoa kun kytkimelle saapuu datakehys ja laskee palaute-funktion jonon koon perusteella. Jos funktion tuottama palaute on negatiivinen, kytkin toimittaa lähettäjälle palautteen tietyllä ennaltamääritellyllä todennäköisyydellä ja lähettäjä vastaanottaessaan palautteen hiljentää lähetysnopeutta palautteeseen perustuen. Jos palaute on positiivinen, kytkin ei lähetä lähettäjälle mitään. Jos lähettäjä ei saa palautetta, lähettäjä kasvattaa lähetysnopeuttaan lisäysalgoritmin mukaan jossa on kolme vaihetta.

Kytkin vastaanottaessaan datakehyksen laskee palautteen

$F_b=(Q_{eq}-Q)-w*(Q-Q_{old})$ jossa Q on kytkimen puskurin pakettijonon koko sillä hetkellä, Q_{eq} on tavoiteltu pakettijonon koko, Q_{old} edellisen palautteen lähetyksen aikana ollut jonon koko ja w on painotettu vakio. Jos $F_b<0$, kytkin toimittaa sen lähettäjälle tietyllä todennäköisyydellä. Edellä olevan palautteen laskevan yhtälön oikealla puolella ensimmäinen suluissa oleva komponentti kertoo kuinka paljon jonon koko eroaa tavoitellusta jonon koosta ja jälkimmäinen suluissa oleva komponentti kertoo kuinka paljon jonon koko on vähentynyt siitä mitä se oli edellisen kerran kun lähettäjä vastaanotti palautteen.

Lähettäjä säätelee lähetysnopeuttaan neljän muuttujan perusteella joita CR (current rate; tämän hetkinen nopeus), TR (target rate; tavoiteltu nopeus), tavulaskuri (byte counter) ja aikalaskuri (time counter). Tavulaskuria kasvatetaan yhdellä kun lähettäjä on lähettänyt tietyn ennalta määrätyn datamäärän *B* ja aikalaskuri kasvaa yhdellä kun tietty ennalta määrätty ajanjakso (15 ms) on kulunut. TR on CR:n tavoiteltu arvo ja $TR \geqslant CR$ aina.

Kun lähettäjä on saanut palautteen, lähettäjä pienentää lähetysnopeutta ja sekä tavulaskuri että aikalaskuri nollataan;

$TR=CR$; $CR=(1-G_d*|F_b|)*CR$ jossa G_d on vakio s.e. $G_d*|max(F_b)|=5$.

Tämän jälkeen nopeutta kasvatetaan kolmessa vaiheessa.

Ensimmäinen vaihe on nopea palautuminen (fast recovery). CR:aa kasvatetaan nopeasti (jos uutta palautetta ei vastaanoteta). Heti kun tavu- tai aikalaskuri on kasvanut yhdellä

CR saa arvon $CR=\frac{CR+TR}{2}$.

Kun joko tavu- tai aikalaskuri saavuttaa arvon 5, alkaa additiivisen lisäyksen vaihe. Tässä vaiheessa aina tavu- tai aikalaskurin kasvaessa yhdellä TR ja CR muutetaan seuraavasti:

$TR = TR + R_{AI}$; $CR = \frac{(CR + TR)}{2}$, missä R_{AI} on vakio (5 Mbps). Tässä vaiheessa tavulaskuri kasvaa yhdellä jo kun puolet *B*:stä on lähetetty ja aikalaskuri laskee yhdellä jo kun 7.5 ms on kulunut.

Kolmas vaihe on hyperaktiivisen lisäyksen vaihe joka alkaa jos edellisen vaiheen jälkeen uutta palautetta ei ole saatu mikä ilmaisee että kaistaa on runsaasti vapaana käytettäväksi. Tällöin TR ja CR lasketaan kuten aktiivisen lisäyksen vaiheessa mutta R_{AI} saa suuremman arvon (50 Mbps) jotta vapaa kaista kulutettaisiin nopeammin.

QCN:aa voidaan soveltaa monilähetyksen ruuhkanhallintaan mutta sellaisenaan voi johtaa siihen että monilähettävien datavirtojen läpisyöttö on pienempi kuin saman pullonkaulalinkin kautta kulkevien yksipolkuisten datavirtojen. Kun samanaikaisesti sekä yksipolkuiset että monilähettävät datavirrat hyödyntävät kytkimen kaistaa, kytkin lähettää palautetta huomattavasti vähemmän monilähettäviltä datavirroilta suhteessa yksipolkuisiin datavirtoihin. Tähän Tanisawa et al. [THY12] esittävät QCN-muunnoksen nimeltä QCN/BS (QCN with Bottleneck Selection) mikä muuttaa datan lähettäjän algoritmia. Lähettäjä ylläpitää taulukkoa jossa on olio jokaista kytkintä kohden joka sisältää senhetkisen lähetysnopeuden *CR*, kohdenopeuden *TR*, sekä tavu- ja aikalaskurit. Lähettäjä tallentaa kaikkien ruuhkautuneiden kytkinten (jotka ovat lähettäjän monilähetettyjen datavirtojen vastaanottajia) lähetysnopeudet ja asettaa lähetysnopeudekseen hitaimman näistä.

Ruan et al. [RWJ16] sanovat että QCN ei hyödynnä linkkien kaistaa tarpeeksi koska QCN:ssä lähettäjä lähettää aluksi linkin maksiminopeudella mutta ei palautteen saamisen ja nopeuden pienentämisen jälkeen pyri hyödyntämään tarpeeksi tehokkaasti vapaaksi jäävää kaistaa. Kirjoittajat esittävät Fast and Simple QCN (FCQCN)-modifikaation QCN:ään jossa kytkimet voivat lähettää lähettäjälle palautteen lisäksi nopeuden resetoivan paketin joka kertoo lähettäjälle että sen täytyy alkaa lähettää suurimmalla mahdollisella lähetysnopeudella. Tällä pyritään hyödyntämään vapaa kaista nopeammin.

5.15 DCQCN

Microsoftin datakeskukset ovat alkaneet käyttää Remote Direct Memory Access-teknologiaa jonka pohjalle Microsoft on kehittänyt Ethernetin ruuhkanhallinta-algoritmin DCQCN joka perustuu QCN:ään ja DCTCP:hen [ZEF15, ZGM16]. Modernit datakeskusverkot hyödyntävät RDMA:ta käyttämällä RDMA Over Converged Ethernet V2-standardia (RoCEv2).

DCQCN perustuu lähetysnopeuden säätelyyn ja suurin osa DCQCN:n toiminnallisuudesta toteutetaan NIC:eissa (network interface controller). QCN:sta poiketen DCQCN:ssa myös datan vastaanottaja on mukana ruuhkanhallinnassa. Kytkimen ajama algoritmi on sama kuin DCTCP:ssa: jos paketin saapuessa kytkimen puskurin jonon koko ylittää kynnysarvon, paketti merkitään. Kun merkitty paketti saapuu vastaanottajalle se lähetetään takaisin lähettäjälle RoCEv2-standarin erityisesti määritettelemänä CNP (Congestion Notification Packet)-pakettina. Lähettäjä säätää lähetysnopeuttaan riippuen siitä onko lähettäjä vastaanottanut CNP-pakettia tietyn ajan sisällä. Jos merkitty paketti saapuu lähettäjälle ja datavirralta johon paketti kuuluu ei ole lähetetty lähettäjälle CNP-pakettia viimeisen N mikrosekunnin aikana, uusi CNP lähetetään välittömästi. Tämän jälkeen joka N mikrosekunnin ajanjakson aikana lähetetään lähettäjälle korkeintaan yksi CNP-paketti (jos tämän ajanjakson aikana on saapunut merkittyjä paketteja). Tämän taustalla motiivina on se että merkityn paketin prosessointi ja CNP:n generointi on kallista minkä vuoksi lähetettyjen CNP-pakettien määrä minimoidaan. Vastaanottaessaan CNP:n, lähettäjä asettaa tavoitellun nopeuden R_T arvoksi tämän hetkisen nopeuden R_C sekä päivittää lähetynopeutta vähentävän muuttujan a DCTCP:n tavoin:

$R_T = R_C$, , $R_C = R_C(1-(\frac{a}{2}))$ ja $a = (1-g)a + g$. R_T saa arvon R_C myöhempää palautumista varten. Jos vastaanottajalle ei ole saapunut merkittyjä paketteja, se ei lähetä CNP:ta lähettäjälle. Jos K aikayksikön aikana lähettäjä ei ole vastaanottanut CNP:ta, a saa arvon $a = (1-g)a$. Kuten QCN:ssä, DCQCN käyttää tavu- ja aikalaskureita kasvattaessaan lähetysnopeutta; aina kun tavulaskuri on kasvanut tietyt B tavua tai ajastin on kasvanut tietyn ajan T verran, lähetysnopeutta kasvatetaan. Muuttujien B ja T säätämisellä voidaan vaikuttaa siihen kuinka tehokkaasti datavirta nostaa lähetysnopeuttaan sen jälkeen kun nopeus on tippunut matalaan arvoon.

Nopeutta kasvatetaan kolmessa vaiheessa (ilman TCP:lle ominaista hidasta aloitusta) jotka ovat kuten QCN:ssä nopea palautuminen, additiivinen lisäys ja hyperaktiivinen lisäys. Nopeassa palautumisessa lähetysnopeus kasvaa $R_C = \frac{(R_T + R_C)}{2}$ kunnes tavu- tai aikalaskuri on kasvanut viiden yksikön verran jonka jälkeen siirrytään additiivisen lisäyksen vaiheeseen jossa lähetysnopeus lähestyy hitaasti tavoiteltua nopeutta: $R_T = R_T + R_{AI}$, $R_C = \frac{(R_T + R_C)}{2}$. Lähettäjät lähettävät aluksi maksiminopeudella (ilman hidasta aloitusta) tehokkuuden vuoksi koska voi olla että kytkimellä ei ole ollenkaan ruuhkaa.

6 Datakeskusten ruuhkanhallinta-algoritmien vertailu

Tässä luvussa vertaillaan edellisessä luvussa esiteltyjä ruuhkanhallinta-algoritmeja. Algoritmit on suunniteltu ajatellen esimerkiksi erityistä ruuhkaa, tietyn tyyppistä liikennettä, erityisiä laitteistovaatimuksia tai erityistä topologiaa ajatellen eikä niiden joukossa ole varsinaista yleispätevää ratkaisua siihen miten datakeskuksen ruuhkanhallinta tulisi toteuttaa. Näin ollen sen sijaan että ratkaisuista ottaisi esille ns. voittajan, tässä vertailussa tuodaan esiin algoritmien eri ominaisuuksia etuineen ja ongelmineen.

Sreekumari ja Jung tekevät laajahkon vertailun datakeskusten eri kuljetusprotokollista [SJ15]. Vertailussa on 25 protokollaa ja ne on jaoteltu kolmeen kategoriaan sen mukaan mitä ongelmaa ne pyrkivät ratkaisemaan (incast, outcast ja latenssi). Kirjoittajat jaottelevat vertailtuja protokollia sen mukaan miten ne pystyvät (tai eivät pysty) kontrolloimaan kolmenlaista aikakatkaisua (aikakatkaisujen on todettu olevan olennaisin syy incastiin). Outcastia torjuvista algoritmeista mainitaan vain 3 joista tässä tutkielmassa on esitelty TCP-CWR sekä DCTCP. Sreekumarin ja Jungin vertaillessa latenssin vähentämiseen pääasiallisesti tähtääviä protokollia on otettu vertailukohdiksi miten protokollat muokkaavat TCP:ta, kytkimiä, ECN:ää, miten paljon ne tarvitsevat tietoa datavirtojen koosta ja ottavatko protokollat huomioon datavirtojen aikarajat. Kirjoittajat toteavat että aikarajat perustuvat yleisesti ohjelmistojen käyttäjäkyselyihin joten varsinaista oikeaa kriteeriä aikarajojen määrittämiselle ei ole.

Zhu et al. [ZGM16] vertailevat DCQCN:ää ja TIMELY:a pohtien kysymystä kumman laiset ruuhkanhallinta-algoritmit ovat toimivampia, ECN:ään perustuvat (DCQCN) vai latenssiin perustuvat (TIMELY). DCQCN:än ja TIMELY:n kehittämisen taustalla on se että pilvidatakeskukset ovat viime aikoina alkaneet hyödyntää RDMA (Remote Direct Memory Access)-teknologiaa joka mahdollistaa huomattavasti suuremman kaistan käytön ja pienemmän latenssin kuin perinteinen TCP/IP. Kirjoittajat päättelevät että TIMELY:n kaltainen protokolla joka käyttää ainoastaan latenssia ruuhkan signaalina voi saavuttaa joko reilun kilpailun datavirtojen kesken tai vakituisen latenssin mutta ei molempia. Artikkelin simulaatioiden perusteella TIMELY ei pysty pitämään kytkimen puskurin jonon pituutta vakaana vaan se vaihtelee suuresti; TIMELY ja DCQCN kuitenkin hyödyntävät kaistan kuta kuinkin yhtä tehokkaasti ja molemmissa latenssi kasvaa samalla kun samanaikaisten datavirtojen määrä kasvaa. Simulaatioissa lyhyiden datavirtojen suoritusaika on TIMELY:ssa isompi kuin DCQCN:ssä jolle kirjoittajat esittävät yhtenä syynä että latenssiin perustuvat ruuhkanhallinta-algoritmit kärsivät ruuhkasignaalin viiveestä; ECN:ään perustuvissa algoritmeissa ruuhkasignaali viivästyy linkkien etenemisviiveen takia mutta latenssiin perustuvissa algoritmeissa ruuhkasignaali myöhästyy sekä etenemisviiveen että pakettien jonotusviiveen vuoksi. Toisena syynä kirjoittajat sanovat että latenssiin perustuvassa ruuhkanhallinnassa vakituisen pakettijonon ylläpito johtaa siihen että kaista ei jakaudu tasaisesti eri datavirtojen kesken. Kolmantena syynä esitetään että kun ruuhkasignaalin viive vastaanottajan ja lähettäjän välillä vaihtelee, tämä vaihtelu vaikuttaa latenssiin perustuviin protokolliin enemmän kuin ECN:ään perustuviin protokolliin; latenssiin perustuvissa protokollissa ruuhkasignaalin viiveen vaihtelu aiheuttaa enemmän häiriöitä itse ruuhkasignaaliin. TIMELY ei edellytä kytkinten modifiointia; kuitenkin se olettaa että verkko pystyy tarkasti mittaamaan viiveet ja pystyy lähettämään ACK:it nopeasti (yleensä viiveiden mittauksissa on tyypillisesti epätarkkuutta johtuen häiriöistä joita tuottavat mm. järjestelmän keskeytykset ja käyttöjärjestelmän prosessointi). Tämän vuoksi TIMELY soveltuu vain verkkoihin joissa verkkosovittimet ovat näin pitkälle kehittyneitä.

DCTCP on yksi yleisimmistä ruuhkanhallinta-algoritmeista joka on otettu käyttöön datakeskuksissa. Kun DCTCP-lähettäjien määrä kasvaa tiettyyn rajaan asti on todettu että incast tapahtuu joka tapauksessa jolloin DCTCP:n läpisyöttö putoaa tavallisen TCP:n tasolle [AM15]. DCTCP:ssä on todettu olevan liian suuri uudelleenlähetyksen

ajastin ja liian suuri alustava lähetysikkunan koko. DCTCP:n käyttöönotto vaatii sekä lähettäjän, kytkinten että vastaanottajien koneiden modifiointia mikä on yleisesti kallista ja työlästä. DCTCP:n etuna voidaan mainita että se kestää hyvin pitkäkestoisia datavirtoja.

Zafar et al. [ZBC16] vertailevat DCTCP:n soveltuvuutta incastin ja outcastin torjuntaan fat-tree- sekä three-tier-topologioissa ja toteavat että DCTCP:n tehokkuus datakeskuksen ruuhkanhallinnassa riippuu suuresti alla olevasta topologiasta. Zafarin simulaation perusteella DCTCP tomiii huomattavasti paremmin fat-tree-topologiassa kuin three-tier-topologiassa. Fat-treessa DCTCP toimii hyvin koska fat-tree-verkossa kaikki kytkimet ovat tyypillisiä halpoja kytkimiä ja DCTCP käyttää tehokkaasti verkon kytkimiä ja linkkejä ruuhkan helpottamiseksi. Lisäksi verkon koon kasvaessa DCTCP toimii paremmin fat-treessa kuin three-tierissa. Samassa tutkimuksessa todetaan myös että DCTCP voittaa TCP:n suurissa verkoissa mutta pienissä verkoissa TCP suoriutuu yhtä hyvin kuin DCTCP. DCTCP:n käyttö pienentää tiedonsiirron viivettä paremmin kuin se onnistuu kasvattamaan läpisyöttöä joten tällä perusteella DCTCP sopii ohjelmille joiden toiminnalle pieni latenssi on kriittisen tärkeätä. Samassa tutkimuksessa havaitaan että incast ja outcast ovat vakavampia räkin sisäisessä liikenteessä ja three-tier- topologiassa.

ICTCP:n käyttöönotto on sikäli edullista että ICTCP:ssä vain vastaanottajan konetta pitää modifioida. ICTCP:n tehokkuus kuitenkin rajoittuu tilanteeseen jossa ruuhkautunut pullonkaulalinkki on viimeinen linkki vastaanottajan ja lähettäjien välillä.

D^2TCP soveltaa DCTCP:ta mutta ottaa tämän lisäksi huomioon datavirran aikarajan laskemalla joka datavirran kohdalla muuttujan joka kertoo kuinka lähellä ajallisesti datavirralle asetettu aikaraja on. D^2TCP ei kuitenkaan ota huomioon muita (mahdollisesti) ruuhkautuneita datavirtoja, toisin sanoen datavirrat eivät tiedä toisistaan mitään eivätkä näin ollen mahdollisesta ruuhkasta. D^2TCP:ssa on sama ongelma skaalautuvuuden kanssa kuin DCTCP:ssa eli datavirtojen määrän lisääntyessä incast tapahtuu joka tapauksessa. UCCS on myös deadlineista tietoinen ratkaisu incastiin joka (toisin kuin D^2TCP) ottaa huomioon kaikki ruuhkautuneeseen pullonkaulalinkkiin osuvat datavirrat priorisoimalla nämä; priorisointi perustuu datavirran aikarajaan ja datavirran kulkeman matkan pituuteen. Toisin kuin D^2TCP, UCCS ei käytä ECN:ää mutta joutuu kuitenkin tallentamaan tietoa jokaisesta ruuhkautuneesta datavirrasta.

UCCS on erityisesti suunniteltu Fat tree-topologiaa ajatellen.

Pfabric on myös aikarajoista tietoinen algoritmi joka on hyvin yksinkertainen idealtaan: kytkimelle saapuvat paketit pudotetaan tai lähetetään eteenpäin datapakettiin ohjelmoidun prioriteetin perusteella ja kyseinen prioriteetti voidaan määritellä miten halutaan (esim. aikaraja, datavirran koko tai datavirran vielä lähettämättä oleva osuus koko datavirran koosta). Pfabricissa kytkimen puskurin koon riittää olla huomattavasti pienempi kuin kytkinten puskurien koko yleensä datakeskuksissa. Koska pFabric perustuu prioriteettijonoille, skaalautuvuus saattaa kuitenkin olla ongelma sillä kun datavirtojen määrä kasvaa ja samalla yhä suurempi joukko prioriteetteja joudutaan asettamaan eri datavirroille (koska aina kun uusi datavirta lisätään verkkoon niin tulee myös lisätä uusi prioriteetin numeerinen arvo, toisin sanoen prioriteettien numeroavaruus kasvaa), on paha sanoa mihin asti datavirtojen (ja siis samalla prioriteettien) määrää voidaan kasvattaa niin että datavirtojen skedulointi pysyisi vielä tehokkaana [MV13].

Tässä tutkielmassa esitellyistä algoritmeista ainakin TCP-CWR ja DCTCP pystyvät torjumaan outcastia [SJ15]. Mekanismeja jotka auttavat outcastin torjumisessa lueteltiin outcastia käsittelevässä luvussa (3.4). Qin et al. [QSS13] toteavat että outcastin kahtena olennaisimpina syinä ovat datavirtojen, joilla on eri tiedonsiirtoviiveet, epätasainen jakautuminen pullonkaulalinkillä sekä datakeskuksissa suoritettavien ohjelmistojen tyypilliset ominaisuudet. TCP-CWR:n etuna on se että vastaanottajan pyytäessä uuden datalohkon kaikkien lähettäjien ruuhkaikkunat ovat aluksi saman kokoisia mikä takaa linkin resurssien reilun jakamisen ainakin tiettyyn pisteeseen asti.

M21TCP-A pyrkii torjumaan incastin asettamalla lähettäjien ruuhkaikkunan koolle maksimin jotta kytkinten puskurit eivät vuotaisi yli. M21TCP-A ei periaatteessa ole ruuhkanhallinta-algoritmi vaan apumekanismi joka varmistaa ettei lähettäjän säätämä ruuhkaikkuna (käytti lähettäjä ruuhkaikkunan säätämiseen mitä algoritmia hyvänsä) ei ylitä tiettyä maksimia. M21TCP-A:n toimivuus rajoittuu yhden polun topologioihin.

MPTCP on ratkaisu topologioihin joissa koneiden välillä on vaihtoehtoisia polkuja (esim. Fat-tree ja Bcube). MPTCP:n linked increase-algoritmi [RWH] on suunniteltu adaptiivisesti tasaamaan kuormaa rinnakkaisten polkujen välillä tehostavan näin verkon läpisyöttöä ja linkkien hyväksikäyttöä. Linked increase, jonka tarkoitus on jakaa verkon polkujen kuormitusta useiden vaihtoehtoisten polkujen kesken, on suunniteltu alunperin

Internetin käyttöön eikä ota huomioon läpisyötön ja latenssin välillä tehtyä kompromissia joka on yksi suurimpia haasteita datakeskusten ruuhkanhallintaa toteuttaessa. Vaikka lähetettävä data jaettaisiin useammaksi alidatavirraksi, tämä ei takaa ettei verkon linkistä tule pullonkaulaa jos kaikki alidatavirrat osuvat samalle linkille. Tämän vuoksi MPTCP:n suoritusteho voi laskea yksipolkuisen TCP:n tasolle kun linkit ovat erityisen ruuhkautuneita, huolimatta siitä että yhden MPTCP-yhteyden alidatavirrat kulkisivat eri polkuja pitkin. Li et al. [LTT14] tutkivat MPTCP:ta incastissa ja toteavat että vaikka MPTCP voi torjua incastia niin MPTCP:n tehokkuutta incastissa rajaa se kuinka monta verkkosovitinta jokaisella verkon palvelimella on (ja verkkosovittimien määrän kasvattaminen tietysti lisää verkon laitteistokustannuksia). Raiciu et al. tutkimuksissa [RPB11] verrataan tavallista TCP:ta ja MPTCP:ta kolmessa eri topologiassa. Eri topologioissa tarvitaan eri määrä alidatavirtoja MPTCP-yhteyttä kohden jotta saavutettaisiin riittävä läpisyöttö. Jotta incast torjuttaisiin lopullisesti, tarvitaan kuitenkin enemmän datavirtoja ja tämän edellyttämät laitteistokustannukset voivat olla suuria. MPTCP on yleisesti kallis ja monimutkainen toteuttaa laitteistotasolla. MPTCP:n hyödyntämisen on kuitenkin havaittu edesauttavan uudenlaisten topologioiden rakentamista.

OTCP käyttää keskitettyä SDN-kontrolleria kerätäkseen tietoa verkon ominaisuuksista ja asettaakseen verkon parametreja sopiviksi. Kerättyjen tietojen perusteella säädetään uudelleenlähetyksen ajastimet ja ruuhkaikkunan alustava koko sekä suurin mahdollinen koko. SDN:n avulla datakeskus-verkon ylläpitoon liittyvä liikenne ja varsinainen ohjelmistojen generoima liikenne erotetaan toisistaan. Jokaisella kytkimellä on erityinen portti millä muodostetaan yhteys ylläpidosta vastaavaan keskitettyyn kontrolleriin. Tämä voi lisää verkon taloudellisia kustannuksia mutta tämän avulla ylläpidon ja ohjelmistojen generoima liikenne eivät häiritse toisiaan. Tätä skenaariota saattavat kuitenkin haitata laitteistoviat ja väärästä parametrien konfiguraatiosta johtuvat ongelmat.

6.1.1 Virtuaalinen ruuhkanhallinta

Cronkrite-Ratcliff et al. [CBV16] pohtivat ongelmaa mikä liittyy virtuaalisiin datakeskuksiin joissa datakeskusten käyttäjät vuokraavat datakeskuksesta palvelimia omaan käyttöönsä ja toteuttavat niissä omia ohjelmistojaan virtuaalikoneiden kautta ja

omia ruuhkanhallinta-protokolliaan.

Näitä ohjelmistoja ei aina ole mahdollista päivittää esimerkiksi uusiin käyttöjärjestelmiin ja näin ollen ne eivät pysty välttämättä ottamaan käyttöönsä uusia ruuhkanhallina-algoritmeja joita virtuaalisessa datakeskuksessa on saatavilla. Ongelma syntyy kun ohjelmistoa ajetaan virtuaalisessa käyttöjärjestelmässä ohjelman käyttäessä esimerkiksi vanhaa TCP-algoritmia joka ei tue ECN:ää. Jos muut ohjelmistot ajavat omia kehittyneempiä ruuhkanhallinta-algoritmejaan, vähemmän kehittyneempää ruuhkanhallintaa käyttävä ohjelmisto ei pysty yhtä tehokkaasti ajamaan tietoliikennettään kuin muut. Kirjoittajat esittelevät virtualisoiduksi ruuhkanhallinnaksi kutsutun idean jossa käyttäjien omat ruuhkanhallinta-algoritmit muunnetaan virtuaalisen datakeskuksen omaksi ylemmän tason ruuhkanhallinta-algoritmiksi (tekstissä käyttäjien omista algoritmeista käytetään nimitystä underlay ja datakeskuksen omasta algoritmista nimitystä overlay). Tämä on mahdollista datakeskuksen hypervisorin avulla jonka kautta kaikki datakeskuksen liikenne kulkee. Ideana on että käyttäjät jotka esimerkiksi käyttävät ECN:ää tukematonta ruuhkanhallintaa pystyvät käyttämään ECN:aa tukevaa algoritmia virtualisoinnin avulla. Hypervisor toteuttaa tämän modifioimalla käyttäjien datapakettien kenttiä sopivalla tavalla.

Protokolla	**TCP-muutokset**	**Algoritmin perusidea**	**Huomioi aikarajat**	**ECN-tuki**	**Ratkaisee**
DCTCP	Lähettäjä, vastaanottaja, kytkin	ECN-palaute + ruuhkaikkunan säätö	Ei	Kyllä	Incast, Outcast, latenssi
ECN*	Kytkin	Muunnettu ECN-kytkimen logiikka	Ei	Kyllä	Incast
ICTCP	Vastaanottaja	Vastaanotto-ikkunaan perustuva	Ei	Ei	Incast
pFabric	Kytkin	Datapakettien priorisointi	Kyllä	Ei	Latenssi
D^2TCP	Lähettäjä, vastaanottaja, kytkin	DCTCP + aikarajan huomiointi	Kyllä	Kyllä	Incast
TCP-CWR	Lähettäjä, vastaanottaja	Lähettäjien ruuhkaikku-	Ei	Ei	Outcast

		noiden koko asetetaan samaksi			
TIMELY	Lähettäjä	Perustuu tarkkoihin tiedonsiirto-viiveiden mittauksiin	Ei	Ei	Incast
MPTCP	Lähettäjä	Data lähetetään useita vaihoehtoisia polkuja pitkin	Ei	Ei	Incast, load balancer
XMP	Lähettäjä	Suuret datavirrat käyttävät MPTCP:ta ja pienet TCP:ta	Ei	Kyllä	Incast, load balancer
DCMC	Lähettäjä, vastaanottaja	Lähetysno-peudeksi valitaan hitaimman DCTCP-yhteyden nopeus	Ei	Ei	Monilähet-tävien ja yksipolkuis-ten datavirtojen (DCTCP) balanssi
M21TCP-A	Lähettäjä, vastaanottaja, kytkin	Lähettäjä laskee maksimin ruuhkaikkun alle	Ei	Ei	Incast
UCCS	Lähettäjä	Datavirtojen priorisointi polun pituuden ja aikarajan perusteella	Kyllä	Ei	Incast
OTCP	Lähettäjä	Keskitetty SDN-kontrolleri säätää ruuhkaikku-naa ja uudelleenlä-hetyksen	Ei	Ei	Incast

		ajastimia			
QCN/BS	Lähettäjä, kytkin	Lähetysnopeus säädetään kytkimen palautteen perusteella	Ei	Ei	Monilähettävien ja yksipolkuisten datavirtojen balanssi
DCQCN	Lähettäjä, vastaanottaja, kytkin	DCTCP + QCN (tietyin muutoksin)	Ei	Kyllä	Incast

Taulukko 6: Datakeskusten ruuhkanhallinta-algoritmeja; taulukko noudattaa osittain Sreekumarin ja Jungin [SJ15] ruuhkanhallinta-algoritmien vertailussa käyttämiä taulukkoja.

7 Yhteenveto

Tässä tutkielmassa tehtiin yleiskatsaus datakeskusten ruuhkanhallintaan. Datakeskuksissa topologiat ja tietoliikenteen ominaisuudet eroavat suuresti Internetin kaltaisista verkoista. Vuosien varrella on havaittu että Internetissä käytetyt tietoliikenteen ratkaisut eivät ole tehokkaita jos niitä sovelletaan suoraan sellaisenaan datakeskuksiin; sen sijaan on tutkittu mekanismeja jotka sopivat yksinomaan datakeskuksen käyttöön. Kuljetus- ja ruuhkanhallinta-protokollat ovat yksi osa-alue joka vaatii erityistä soveltamista datakeskusten kaltaisissa ympäristöissä. Internetin kuljetusprotokolla TCP ei ole tarpeeksi tehokas datakeskuksissa joille tyypillistä ovat korkea läpisyöttö ja pienet latenssit.

TCP Incast on tilanne jossa asiakaskone pyytää dataa jonka osia säilyttävät samanaikaisesti useat koneet ympäri verkkoa ja näiden koneiden vastatessa asiakkaalle samanaikaisesti lähettäjien ja vastaanottajan välissä oleva linkki ylikuormittuu. Incast johtaa runsaaseen datapakettien pudottamiseen ja toistuviin uudelleenlähetyksiin mikä voi heikentää verkon läpisyöttöä radikaalisti. TCP Outcast on taas tilanne jossa erikokoiset datavirrat yrittävät samaan aikaan kulkea saman linkin läpi jolloin linkin tarjoama kaista jakautuu epätasaisesti eri datavirtojen kesken.

Suurille datavirroille on yleensä tärkeää korkea läpisyöttö kun taas pienille datavirroille on tärkeää pieni latenssi. Suuret datavirrat ovat yleensä järjestelmän ylläpidosta syntyviä datavirtoja ja pienet datavirrat liittyvät ohjelmistoihin jotka pyrkivät

tarjoamaan nopean vastauksen käyttäjälle. Koska suuret datavirrat kyllästävät helposti verkon linkit hidastaen pienten datavirtojen etenemistä, haaste datakeskusten ruuhkanhallinnassa onkin tasapainotella sopivasti suurien ja pienien datavirtojen vaatimusten välillä, toisin sanoen korkean läpisyötön ja latenssin välillä.

Ratkaisuja datakeskusten ruuhkanhallintaan voidaan vertailla monin perustein, kuten sen mukaan minkälaisia laitteistoratkaisuja ne edellyttävät ja minkälaisiin topologioihin ruuhkanhallinta on tarkoitettu. Monesti ruuhkanhallintaan esitetyt ratkaisut ovat erityisen spesifejä olosuhteisiin nähden, toisin sanoen ne on esitetty esimerkiksi tietyn tyyppistä liikennettä, topologioita ja ohjelmistoja ajatellen.

Lähteet

AGM10 Alizadeh, M. et al., Data center TCP (DCTCP). Proc. of the ACM SIGCOMM 2010 conference. New Delhi, India, 30.8 - 3.9.2010.

AJP11 Alizadeh, M., Javanmard, A. & Prabhakar, B., Analysis of DCTCP: Stability, Convergence and Fairness. ACM SIGMETRICS Performance Evaluation Review - Performance evaluation review, Vol. 39, Nro. 1, 2011 (kesäkuu), s. 73-84.

AM15 Adesanmi, A. & Mhamdi, L., Controlling TCP Incast Congestion in Data Center Networks. International Conference on Communication Workshop, IEEE, 8.6-12.6 2015.

AMY16 Akamatsu, J., Matsushima, K. & Yamamoto, M., Equation-based Multicast Congestion Control in Data Center Networks. Proc. on 2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS), 5.10-7.10 2016.

AVS04 Albuquerque, C., Vickers, B. J. & Suda, T., Network Border Patrol: Preventing Congestion Collapse and Promoting Fairness in the Internet. IEEE/ACM Transactions on Networking, vol. 12, nro 1, helmikuu 2004.

AYS13 Alizdeh, M. et al., pFabric: Minimal Near-optimal Datacenter Transport. Proc. of the ACM SIGCOMM 2013 Conference, ACM, Hong Kong, Kiina, 12.8-16.8 2013.

BBR10 Barré et al., Experimenting with Multipath TCP. Proceedings of the ACM SIGCOMM 2010 conference, ACM, New Delhi, India, 30.8. - 3.9. 2010.

BCV Baiocchi, A., Castellani, A. & Vacirca, F., YeAH-TCP: Yet Another High-speed TCP. https://www.researchgate.net/publication/228561173_YeAH-TCP_Yet_another_highspeed_TCP, viitattu 24.4.2017.

BSW14 Bradjonic, M., Saniee, I. & Widjaja, I., Scaling of Capacity and Reliability in Data Center Networks. ACM SIGMETRICS Performance Evaluation Review, vol. 2, nro 2, syyskuu 2014.

CBV16 Cronkrite-Ratcliff, B et al., Virtualized Congestion Control. Proc. of the 2016 ACM SIGCOMM Conference, ACM, Florianopolis, Brasilia, 22.8-26. 2016.

CXF13 Cao, Yu et al., Explicit Multipath Congestion Control for Data Center Networks. Proc. of the 9th ACM Conference on Emerging Networking Experiments and Technologies, ACM, Santa Barbara, California, USA, 9.12-12.12 2013.

GLL09 Guo et al., BCube: A High Performance Server Centric Network Architecture for Modular Data Centers. Proc. of the ACM SIGCOMM 2009 Conference on Data Communication, Barcelona, Espanja, 16.8 - 21.8. 2009.

JA10 Jamali, S. & Analoui, M., Globally stable and high-performance Internet congestion control through a computational inspiration from nature. Science China Information Sciences, 2011.

LJM12 Lee, C., Jang, K. & Moon, S., Reviving Delay-based TCP for Data Centers. Proc. of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures and Protocols for Computer Communication, Helsinki, Suomi, 13.8.-17.8. 2012.

LLT14 Li, M. et al., MPTCP Incast in Data Center Networks. China Communications, vol. 11, nro 4, huhtikuu 2014.

MLD15 Mittal, R. et. al, TIMELY: RTT-based Congestion Control for Data Center. Proc. of the 2015 ACM Conference on Special Interest Group for Data Communication, ACM, Lontoo, Iso-Britannia, 17.8 -21.8 2015.

MV13 Mu, W & Vuong, B., CS244 '13: Pfabric: Deconstructing Data Center Transport. https://reproducingnetworkresearch.wordpress.com/2013/03/13/cs244-13-pfabric-deconstructing-data-center-transport/, viitattu 11.4.2017.

NGS16 Nguyen, T., Gangadhar, S. & Sterbenz, J., Performance Evaluation of TCP

Congestion Control Algorithms in Data Center Networks. Proc. of the 11th International Conference on Future Internet Technologies, ACM, Nanjing, Kiina, 15.6.-17.6. 2016.

OR16 Olteanu, V. & Raiciu, C., Datacenter Scale Load Balancing for Multipath Transport. Proc. of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization, ACM, Florianopolis, Brasilia, 22.8.-26.8 2016.

PB14 Paasch, C. & Bonaventure, O., Multipath TCP. ACM Queue, Vol. 2, Nro. 2, 2014 (maaliskuu).

PDH12 Prakash, P. et al., The TCP Outcast Problem: Exposing Unfairness in Data Center Networks. Proc. of the 9th Usenix Conference on Networked Systems Design and Implementation, Usenix, San Jose, California, USA, 25.4-27.4 2012.

PJP16 Po Sto, F., Jouet, S. & Pezaros, D., Network and Resource Management Strategies for Data Center Infrastructures: a Survey. Computer Networks, vol. 106, 4.9.2016.

PW11 Podlesny, M. & Williamson, C., An Application-Level Solution for the TCP-incast Problem in Data Center Networks. Proc. of the Nineteenth International Workshop on Quality of Service, IEEE Press, 2011.

PWS11 Papadimitrou et al., Open Research Issues in Internet Congestion Control. https://tools.ietf.org/html/rfc6077, viitattu 30.3.2017.

QSS13 Qin, Y. et al., Analysis of Unfairness of TCP Outcast Problem in Data Center Networks. Proc. of the 2013 25th International Teletraffic Congress, 10.9-12.9 2013.

RPB10 Raiciu et al., Data Center Networking with Multipath TCP. Proc. of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, ACM, Monterey, Kalifornia, 20.10. - 21.10.2010.

RPB11 Raiciu et al., Improving Datacenter Performance and Robustness with Multipath TCP. Proc. of the ACM SIGCOMM 2011 Conference, ACM, Toronto, Ontario, Kanada, 15.8-19.8 2011.

RWH Raiciu. C., Wischik, D. & Handley, M., Linked Congestion Control. http://nrg.cs.ucl.ac.uk/mptcp/presentations/mptcp-76-linked-cc.pdf, viitattu 20.2.2017.

RWJ16 Ruan, C., Wang, J. & Jiang, W., FSQCN: Fast and Simple Quantized Congestion Notification in Data Center Ethernet. Proc. on the 36th International Conference on Distributed Computing Systems, IEEE, 27.6-30.6 2016.

SJ15 Sreekumari, P. & Jung, J., Transport Protocols for Data Center Networks, a Survey of Issues, Solutions and Challenges. Photonic Network Communications, vol. 31, nro 1, helmikuu 2016.

SSC16 Souza Couto, R. et al., Reliability and Survivability Analysis of Data Center Network Topologies. Journal of Network and System Management, vol. 24, nro 2, huhtikuu 2016.

THY12 Tanisawa, Y., Hayashi, Y. & Yamamoto, M., Quantized Congestion Notification for Multicast In Data Center Network. Proc. on First International Conference on Cloud Networking (CLOUDNET), IEEE, 28.11-30.11 2012.

TYP16 Tseng, H., Yang, T., & Peng, Y., An urgency and congestion control scheme for larger-scale TCP incast problem in data center. Proc. on the 2015 IEEE Symposium on Computers and Communication, IEEE, 6.7-9.7 2016.

VHV12 Vamanan, B., Hasan, J. & Vijaykumar, T. N., Deadline-aware Datacenter TCP. Proc. of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures and Protocols for Computer Communication, ACM, Helsinki, Suomi, 13.8-17.8 2012.

WFG10 Wu, F. et al., ICTCP: Incast Congestion Control for TCP in data center networks. Co-Next 10, ACM, Philadelphia, Pennsylvania, 30.11-3.12.2010.

WJL12 Wu, H. et al., Tuning ECN for Data Center Networks. Co-NEXT' 12, Nizza, Ranska, 10.12.-13.12. 2012.

ZA13 Zhang, Y. & Ansari, N., On Architecture Design, Congestion Notification, TCP Incast and Power Consumption in Data Centers. IEEE Communications Surveys & Tutorials, vol 15, nro 1, 2013.

ZBC16 Zafar, S., Bashir, A. & Chaudhry, S. A., On Implementation of DCTCP on Three-tier and Fat-Tree Data Center Network Topologies. SpringerPlus, vol. 5, numero 766, kesäkuu 2016.

ZEF15 Zhu, Y. et al., Congestion Control for Large-Scale RDMA Deployments. Proc. of the 2015 ACM Conference on Special Interest Group on Data Communication, ACM, Lontoo, Iso-Britannia, 17.8-21.8 2015.

ZGM16 Zhu, Y. et al., ECN or Delay: Lessons Learnt from Analysis of DCQCN and Timely. Proc. of the 12th International Conference on Emerging Networking Experiments and Technologies, Irvine, Kalifornia, USA, 12.12.-15.12. 2016.

ZXin16 Zhang, X., Adaptive Flow Scheduling for Modular Datacenter Networks. Peer-to-peer Networking and Applications, 2016.